Deux critères de sécurité pour l'exécution de code mobile

Hervé Grall

Thèse présentée pour obtenir le titre de

Docteur de l'École Nationale des Ponts et Chaussées

Spécialité

Mathématiques – Informatique

par

Hervé Grall

Deux critères de sécurité pour l'exécution de code mobile

Soutenue le 15 décembre 2003 devant le jury composé de :

M. Xavier Leroy (Président)
M. Norbert Cot (Directeur de Thèse)
M. Didier Caucal (Rapporteur)
M. Thomas Jensen (Rapporteur)
M. René Lalement

Les programmes mobiles, comme les applettes, sont utiles mais potentiellement hostiles, et il faut donc pouvoir s'assurer que leur exécution n'est pas dangereuse pour le système hôte. Une solution est d'exécuter le programme mobile dans un environnement sécurisé, servant d'interface avec les ressources locales, dans le but de contrôler les flux d'informations entre le code mobile et les ressources, ainsi que les accès du code mobile aux ressources. Nous proposons deux critères de sécurité pour l'exécution de code mobile, obtenus chacun à partir d'une analyse de l'environnement local. Le premier porte sur les flux d'informations, garantit la confidentialité, est fondé sur le code de l'environnement, et est exact et indécidable ; le second porte sur les contrôles d'accès, garantit le confinement, s'obtient à partir du type de l'environnement, et est approché et décidable.
Le premier chapitre, méthodologique, présente l'étude d'objets infinis représentés sous la forme d'arbres. Nous avons privilégié pour les définir, une approche équationnelle, et pour raisonner sur eux, une approche déductive, fondée sur l'interprétation co-inductive de systèmes d'inférence. Dans le second chapitre, on montre dans le cas simple du λ-calcul, comment passer d'une sémantique opérationnelle à une sémantique dénotationnelle, en utilisant comme dénotation d'un programme son observation. Le troisième chapitre présente finalement en détail les deux critères de sécurité pour l'exécution de code mobile. Les techniques développées dans les chapitres précédents sont utilisées pour l'étude de la confidentialité, alors que des techniques élémentaires suffisent à l'étude du confinement.

Two Security Criteria for Executing Mobile Code

Mobile programs, like applets, are not only ubiquitous, but also potentially malicious. Therefore, the host system must grant that their execution is not dangerous for its resources. A solution is to execute mobile programs in a secured environment, which enables controlling information flows between mobile programs and local resources, and accesses from mobile programs to local resources. We give two security criteria for executing mobile code. The first one deals with information flows, ensures confidentiality, is based on the code of the local environment, is accurate and undecidable ; the second one deals with access controls, ensures confinement, is based on the type of the local environment, is approximate and decidable.

Mots-clés Logique mathématique – Systèmes d'inférence – Définitions inductives et co-inductives – Treillis – Informatique théorique – Équations récursives – Arbres – Lambda-calcul – Langages de programmation – Sémantiques opérationnelle et dénotationnelle – Sécurité informatique – Mobilité – Flux d'informations – Confidentialité – Contrôles d'accès – Confinement

Remerciements

C'est René Lalement qui m'a proposé le sujet de cette thèse, la sécurité d'exécution de code mobile : pour ce choix si judicieux, je le remercie. C'est aussi sous sa direction que j'ai travaillé : tout en me laissant une grande liberté, il a exercé sur mon travail une influence considérable, en particulier par les connaissances qu'il m'a transmises avec une clarté qui me sert désormais de modèle.

Je remercie Norbert Cot qui a accepté de diriger ma thèse, une direction un peu particulière puisqu'entièrement déléguée, sur le plan scientifique, à René Lalement. Ses conseils et ses encouragements m'ont été précieux.

Je suis heureux de remercier ici Xavier Leroy : il a présidé non seulement mon jury de thèse, mais aussi à l'orientation de mon travail, grâce à son papier de POPL '98, écrit avec François Rouaix, et concernant la sécurité des applettes ; sa lecture en première année de thèse a été décisive pour la suite, puisque j'y ai trouvé toute la problématique de ma thèse, mais aussi les questions à résoudre, la conjecture concernant le confinement et la question ouverte portant sur le passage au contexte des analyses de sécurité. J'ai aussi beaucoup apprécié la disponibilité de Xavier Leroy et François Rouaix, lorsqu'ils m'ont reçu pour discuter de leur article.

Je dois beaucoup à Didier Caucal, un intérêt pour les structures infinies que j'avais commencé à étudier en DEA sous sa direction, mais aussi une attitude, une sorte de réflexe du chercheur qu'on ne manque pas de développer à son contact. Je le remercie vivement d'avoir accepté d'être le rapporteur de cette thèse alors même qu'un tiers seulement du mémoire concerne son domaine ; je le remercie aussi d'avoir contribué par ses commentaires à l'amélioration du premier chapitre du mémoire.

Thomas Jensen a accepté d'être rapporteur de ma thèse, après la seule lecture de l'introduction et sans me connaître. De cette confiance, je le remercie chaleureusement. Je lui suis d'autant plus reconnaissant de ses encouragements et de tous ses commentaires et conseils, qui m'inspirent toujours, que je suis conscient de lui avoir imposé une tâche fastidieuse, tant ce mémoire est anormalement long.

J'ai effectué mon doctorat au CERMICS[1], dont je garde un excellent souvenir. Je remercie particulièrement Gilbert Caplain, qui a largement contribué à l'amélioration de la rédaction de cette thèse par ses relectures at-

[1]Centre d'Enseignement et de Recherche en Mathématiques, Informatique et Calcul Scientifique – Laboratoire commun à l'École Nationale des Ponts et Chaussées et à l'Institut National de Recherche en Informatique et Automatique – adresse : ENPC, 6 et 8 avenue Blaise Pascal, Cité Descartes, Champs-sur-Marne, 77455 Marne-la-Vallée Cedex 2

tentives. Je remercie aussi pour leur aide, toujours efficace et sympathique, Bernard Lapeyre, directeur du CERMICS, Jacques Daniel, administrateur du système informatique, Sylvie Berte et Imane Hamade, du secrétariat, ainsi que tous les membres de l'équipe d'informatique, qui était alors composée, outre René Lalement et Gilbert Caplain, de Daniel Hirschkoff, Mathieu Jaume, Renaud Keriven et Thierry Salset.

Je suis reconnaissant à l'école des Ponts et Chaussées d'avoir assuré le financement de mes deux premières années de thèse. Je remercie tous les membres du collège doctoral, en particulier son président, Nicolas Bouleau, ainsi que Claude Tu, Marine Daniel et Alice Tran. En troisième année, j'ai été ATER à l'université de Bretagne-Sud. J'ai beaucoup appris en participant aux cours de Jacques Malenfant et de Frédéric Raimbault ; je garde aussi un agréable souvenir de ma collaboration avec Moncef Daoud et Jérôme Goulian.

Table des matières

Introduction

Le code mobile[1] est utile mais hostile, au moins potentiellement, et il faut donc pouvoir s'assurer que l'exécution de code mobile n'est pas dangereuse pour le système sur lequel elle se produit. Aussi, le système hôte exécute le code mobile dans un environnement sécurisé, formé de code local et servant d'interface avec les ressources de l'hôte, dans le but de les protéger, particulièrement de contrôler

- les accès du code mobile aux ressources de l'hôte,
- les flux d'informations entre le code mobile et les ressources.

Prenons l'exemple d'un environnement contenant une ressource munie d'une fonction d'accès, et supposons que l'accès à cette ressource doive être contrôlé. Deux possibilités se présentent alors : ou bien l'environnement permet un accès direct à la ressource, protégée par le remplacement de la fonction d'accès encapsulée par une version sûre, réalisant les contrôles avant l'accès, ou bien l'environnement propose une interface d'accès à la ressource, composée d'une fonction sûre réalisant les contrôles, et assure que la ressource est confinée dans l'environnement ; en effet, si ce n'était pas le cas, le code mobile pourrait appeler directement la fonction d'accès encapsulée dans la ressource, sans aucun contrôle.
Supposons maintenant que la fonction d'accès lise des informations contenues dans la ressource, et que certaines de ces informations doivent rester confidentielles. Dans ce cas, ce sont les flux d'informations qui doivent être contrôlés, afin d'éviter que le code mobile puisse prendre connaissance d'informations confidentielles.

Nous soutenons la thèse suivante : pour vérifier le confinement de ressources ou déterminer les flux d'informations entre les ressources et le code mobile, il est possible d'analyser l'environnement local seul. Chacune de ces analyses repose sur l'examen de la frontière entre le code mobile et son envi-

[1] On entend par code un ensemble d'instructions écrites dans un langage de programmation. Opposé au code local, attaché à une machine particulière, le code mobile s'exécute sur une machine hôte à laquelle il est lié temporairement, le temps de l'exécution.

ronnement d'exécution. Précisément, dans le cas du confinement, la frontière est définie de manière locale et dynamique, ce qui permet de décrire les interactions entre le code mobile et l'environnement local au cours de l'exécution et de vérifier que les ressources sont confinées dans l'environnement ; dans le cas des flux d'informations, la frontière est définie de manière globale et permanente, ce qui permet de décrire ce qu'observe le code mobile de l'environnement local, puis de repérer les éventuels flux d'informations entre les ressources et le code mobile.

Il est clair que la mobilité introduit une frontière au sein même du code, le partageant entre le code mobile et le code local. C'est la variation la plus simple de l'unité originelle, libre de toute frontière, où du code s'exécute sur une machine[2] ; elle peut se rencontrer en beaucoup de situations, que nous évoquons brièvement.
Le code mobile peut se déplacer sur un réseau, comme Internet. On pense évidemment aux applettes du langage Java, qui sont des applications dont le code compilé peut être attaché à un document du Web ; après avoir chargé un document contenant une applette, un navigateur peut exécuter l'applette sur une machine virtuelle dans un environnement sûr. Concrètement, pour des raisons d'incompatibilité avec certains navigateurs, les applettes sont peu utilisées sur le Web.
Le code mobile peut aussi être chargé sur des composants dédiés à des tâches spécifiques, comme les cartes à puces qui contiennent un processeur et de la mémoire, volatile et permanente. Compte tenu des ressources limitées d'une carte[3], un sous-ensemble de Java a été développé pour les cartes à puces, Java Card. Se trouvent alors sur la carte à puces une machine virtuelle pour Java Card, un environnement sûr, et des applettes, qui sont téléchargées suivant les besoins spécifiques.

Après ces considérations pratiques, venons-en à la construction de l'objet d'étude, ainsi qu'à la problématique.

Le code mobile s'exécute sur une machine hôte en appelant des fonctions présentes dans l'environnement d'exécution hôte, formé de code local. À partir d'analyses de l'environnement seul, on cherche à garantir la sécurité d'exécution de tout code mobile envisageable.

[2]La machine peut être abstraite ou concrète, suivant que le code est exprimé dans un langage de haut niveau ou de bas niveau. Elle peut être virtuelle ou physique, suivant qu'elle est implémentée par logiciel ou par matériel.

[3]La fréquence du processeur est de l'ordre du méga-hertz pour une carte à puces, du giga-hertz pour un ordinateur, la capacité de la mémoire vive de l'ordre du kilo-octet pour une carte à puces, du méga-octet pour un ordinateur.

Dans cette introduction, on s'intéresse particulièrement au langage Java, puisque ce langage, par sa large diffusion et par les possibilités de mobilité qu'il a introduites, a contribué à modeler l'ensemble de la problématique sécuritaire. On se réfère aussi à des travaux portant sur des langages beaucoup plus simples que Java ; il faut bien comprendre que cette simplification concerne avant tout la syntaxe, car il est généralement possible de traduire une partie substantielle de Java en ces langages. Ainsi, en utilisant ces langages simplifiés, on se livre à une modélisation, qui ne retient que les caractéristiques importantes et permet une résolution formelle des problèmes rencontrés. Pour être utile, une modélisation doit permettre en retour d'appliquer la solution à des langages comme Java. Nous reviendrons sur cette question en conclusion, car elle est importante : nous utilisons en effet dans ce travail des langages de programmation très simples, l'un purement fonctionnel, l'autre enrichi par l'ajout de références permettant la manipulation d'objets en mémoire.

Commençons par définir ce qu'on peut entendre par la sécurité d'exécution. Dès qu'un système informatique possède des biens (des ressources) à protéger, la question de la sécurité se pose, et peut être formulée sous la forme d'exigences sécuritaires. Ces exigences peuvent porter

- sur la spécification fonctionnelle du système, en définissant alors une politique de sécurité, ensemble de règles que les fonctions de sécurité du système doivent vérifier,
- de manière associée, sur la démonstration que le système applique effectivement la politique de sécurité, en indiquant le degré de formalisation de cette démonstration,
- enfin, sur les conditions d'utilisation du système.

Dans le cas qui nous intéresse, le système est constitué de la machine et de l'environnement hôtes. Par machine, on entend ici la machine abstraite exécutant le code, précisément sa représentation sémantique ; par environnement, un ensemble de fonctions écrites dans le langage de programmation considéré et formées de code local. On suppose que ce langage est un langage de haut niveau, manipulant des entités abstraites, comme Java. Ainsi, nous partons d'une vue abstraite du système, se fondant sur le langage de programmation utilisé, défini par sa syntaxe et sa sémantique opérationnelle. Ce point de départ n'est pas sans conséquences sur l'implémentation des fonctions de sécurité : elle pourra s'effectuer au niveau du code, en l'instrumentant, ou au niveau du langage, par exemple en modifiant le système de types ou la sémantique.

Venons-en aux exigences, et en premier lieu aux conditions d'utilisation de notre système.

La sécurité du système doit être garantie pour l'exécution de tout code mobile. Cette condition impose que le langage assure une sûreté minimale, celle du typage. Dans ce cas, aucune exécution ne peut provoquer d'erreurs de types, ce qui permet d'éviter par exemple les comportements suivants dans le code mobile, fort compromettants pour la sécurité :

- la conversion d'une chaîne de caractères en une fonction, permettant d'exécuter n'importe quel code,
- la conversion d'une chaîne en une adresse en mémoire, permettant d'accéder à la mémoire de manière incontrôlée.

C'est évidemment une des caractéristiques majeures de Java que de posséder un système de types sûr. Plus généralement, on impose l'utilisation de langages dont le système de types est sûr.

Pratiquement, le code mobile et le code local sont formés de code exécutable (sur une machine virtuelle ou physique), résultat de la compilation du code écrit dans le langage de programmation de haut niveau. Une question se pose donc : notre point de vue abstrait est-il justifié ?

Il l'est si la sécurité, garantie initialement pour l'exécution de tout code mobile compilable appelant le code local compilable, est préservée par compilation : autrement dit, la sécurité est-elle encore garantie pour l'exécution de tout code mobile exécutable appelant le code local exécutable ?

Comme le remarque Abadi [1], la compilation, considérée comme une traduction d'un langage de haut niveau en un langage de plus bas niveau, n'est généralement pas une traduction complètement adéquate[4] : dans ce cas, une propriété vérifiée par du code compilable ne l'est plus nécessairement après compilation. Abadi présente le cas du langage Java et de la traduction en « bytecode », le langage de la machine virtuelle de Java. Par un exemple probant (cf. [1, §2.2]), il montre qu'au niveau du « bytecode », il est possible de distinguer les traductions de deux programmes, pourtant équivalents avant compilation : le « bytecode » permet en effet de créer des contextes qui ne sont pas le résultat d'une compilation ; le « bytecode » est donc un langage trop riche. Ainsi, il peut être nécessaire de vérifier le code mobile exécutable à sa réception, pour montrer qu'il est bien le résultat d'une compilation.

La nécessité d'une vérification s'impose indépendamment du problème de l'adéquation complète : en effet, rien ne permet de supposer que le code mobile à exécuter est légal, autrement dit appartient au langage de la machine virtuelle. Il pourrait violer par exemple des règles de typage, compromettant ainsi la sécurité lors de son exécution. Dans le cas de Java, où le « bytecode » est un langage typé, le code s'exécutant sur la machine virtuelle

[4] « Fully abstract » en anglais.

est vérifié, statiquement et dynamiquement, de manière à assurer l'absence de certaines erreurs à l'exécution : on pourra lire, au sujet des vérifications statiques, l'article de Leroy [47], en particulier le début qui décrit la machine virtuelle, le « bytecode » et les vérifications effectuées qui concernent le typage.
Comment faciliter la vérification de code exécutable ? Comme pour le « bytecode » du langage Java, il est préférable de disposer d'un langage typé. Comme en témoigne par exemple les compte-rendus de l'atelier de travail « Types in compilation » [48], une tendance s'affirme, celle d'utiliser, lors de la compilation, des langages intermédiaires typés. Poussée à sa limite, elle aboutit au langage typé pour assembleur[5], conçu par Morrisett et al. [59] pour assurer :

- la sûreté de la mémoire, soit la correction des accès à la mémoire,
- la sûreté du flux de contrôle, soit l'exécution des instructions suivant le séquencement programmé,

ce qui peut se résumer à la préservation des abstractions. On peut donc considérer que le code mobile appartenant à un tel langage transporte avec lui des informations de typage, permettant de démontrer certaines propriétés de sûreté. C'est un exemple d'une technique plus générale développée par Lee et Necula [60], préconisant l'utilisation de code transportant sa preuve (« Proof Carrying Code ») : à la réception du code mobile, la preuve qu'il transporte est vérifiée, puis si elle est valide, le code est exécuté. Cette technique permet d'étendre les propriétés démontrées au delà de la sûreté du typage.

Par la suite, nous conservons un point de vue abstrait et négligeons le problème de la préservation des propriétés de sécurité par compilation. Nous allons raisonner à partir du code local formant l'environnement et du code mobile, entendus comme des programmes appartenant au langage de haut niveau considéré : on parlera ainsi du programme local, contenant le code local formant l'environnement, et du programme mobile, contenant le code mobile.
On entend ici par programme une entité contenant du code et pouvant s'exécuter. Notre terminologie peut paraître impropre dans le sens où l'environnement est plutôt une bibliothèque de sous-programmes et le code mobile

[5] Jusqu'à maintenant, nous avons décrit la compilation simplement comme la traduction de code compilable en code exécutable sur une machine, virtuelle ou physique. On peut considérer que le code pour assembleur peut être interprété par une machine virtuelle très proche de la machine physique : l'assembleur traduit en effet le code à assembler, qui utilise des instructions symboliques, en code machine, qui utilise des instructions sous forme binaire.

du code pouvant s'exécuter s'il est lié à l'environnement. À vrai dire, un modèle simple permet de rendre compte de cette situation tout en justifiant notre terminologie : le programme mobile est une fonction à un paramètre, et le programme local la valeur effective de ce paramètre. C'est toujours à ce modèle que nous nous référons.
Ce niveau d'abstraction va nous permettre de raisonner formellement, donc de démontrer rigoureusement les propriétés de sécurité : c'est donc l'exigence la plus forte qu'on puisse avoir pour la démonstration que les fonctions de sécurité appliquent correctement la politique de sécurité.

Quelles exigences fonctionnelles de sécurité peut-on prendre en compte ? En appelant l'environnement, le code mobile peut accéder à des ressources du système hôte ou obtenir des informations sur leur contenu : dans la mesure où le code mobile est potentiellement hostile, il est nécessaire de contrôler ces accès et ces flux d'informations, par l'intermédiaire de l'environnement.

Contrôler les flux d'informations La notion de flux d'informations peut être définie de manière générale, comme dans l'article de Sutherland [79]. Considérons un système possédant plusieurs états, formant l'ensemble S, et soumis à l'observation de deux observateurs : chaque observateur associe à chaque état du système une valeur, résultat de l'observation, ce qui permet de définir pour chaque observateur l'ensemble des valeurs observables. Les flux d'informations peuvent se définir indirectement ainsi, à partir de la notion d'indépendance :

> les observations des deux observateurs sont indépendantes si pour toutes valeurs δ_1 et δ_2 observables respectivement par le premier et le second observateur, il existe un état de S dont l'observation par le premier observateur donne δ_1, et par le second, δ_2.

Si leurs observations sont dépendantes, alors il existe deux valeurs observables δ_1 et δ_2 pour lesquelles il n'existe aucun état de S conduisant aux observations respectives de δ_1 et δ_2. Dans ce cas, dans un état donné du système, si le premier observateur observe δ_1, il en déduit que le second observateur n'observe pas δ_2 : il existe donc un flux d'informations du second observateur vers le premier.
À cette condition d'indépendance, il est souvent préféré celle de non-interférence, plus forte :

> les observations du premier observateur n'interfèrent pas avec celles du second si le second observateur n'observe qu'une unique valeur.

Cette définition se comprend mieux si on considère que le système est un programme, acceptant une valeur en entrée, observée par le premier observateur, et produisant un résultat, observé par le second observateur[6] : elle signifie alors que le résultat observable ne dépend pas de la valeur en entrée, autrement dit que l'observation du résultat respecte la confidentialité de la valeur en entrée. Sous cette forme, on retrouve la définition de l'indépendance donnée par Cohen dans [19][7], et celle de la non-interférence donnée par Volpano, Smith et Irvine dans [84], qui reprennent le terme utilisé par Goguen et Meseguer dans [29] pour décrire une propriété de sécurité des systèmes (d'exploitation) multi-utilisateurs.
Cette propriété de non-interférence est importante, car depuis l'article de Volpano et al., elle est utilisée pour justifier la correction des analyses statiques des flux d'informations, présentées généralement sous la forme de systèmes de types, et inspirées peu ou prou du travail fondateur de Denning [26]. Ces analyses traquent les dépendances, ou bien directes, comme dans une affectation à une variable, où le contenu de la variable dépend de la valeur affectée, ou bien indirectes, comme dans une alternative, où les branches dépendent toutes deux de la condition. On peut lire à ce sujet l'article de Myers et Sabelfeld [73] qui recense et classe un ensemble exhaustif de travaux relatifs à l'analyse statique des flux d'informations.
Pour le système qui nous intéresse, composé de la machine hôte et de l'environnement formé de code local, le contrôle des flux d'informations va se traduire par le respect d'une propriété d'indépendance, équivalente à une propriété de non-interférence relative comme celle qui précède à la confidentialité.
On suppose que le programme local, c'est-à-dire l'environnement formé de code local, est paramétré par la valeur d'un sous-programme, représentant une ressource dont le contenu doit rester confidentiel : c'est la valeur de ce paramètre local qui va être observée initialement. Ainsi, un état du système peut être décrit par :

- la valeur initiale du paramètre local,
- le programme mobile, formé du code mobile, qui s'exécute en appelant le programme local.

[6] On peut observer d'un résultat sa valeur, mais pas toujours. Les valeurs des types de base, dits « passifs », comme les entiers, les booléens, sont observables. Au contraire, les valeurs des types « actifs », comme les types des fonctions ou des objets, ne le sont pas : l'observation se limite alors à la terminaison du programme. Par la suite, on entend par résultat observable ce qu'on peut observer du résultat.

[7] Cette référence, peu connue, est citée dans l'article de Sabelfeld et Sands [74], d'où nous la reprenons.

D'un état on observe :

- d'un point de vue local, la valeur initiale du paramètre local,
- d'un point de vue extérieur, l'exécution du code mobile dans l'environnement local, par exemple en n'en retenant que la valeur du programme mobile et son résultat d'exécution.

La propriété de sécurité à vérifier concerne la confidentialité du contenu de la ressource, et s'exprime donc simplement par l'indépendance entre les observations des valeurs du paramètre local et celles des exécutions. Clairement, elle est équivalente à la propriété suivante, utilisant la non-interférence :

> pour tout programme mobile, les observations des valeurs du paramètre local n'interfèrent pas avec les observations des exécutions du programme mobile appelant le programme local.

Si l'on n'observe d'une exécution que son résultat final, la propriété de confidentialité peut donc s'exprimer sous la forme suivante, par définition de la non-interférence :

> pour tout programme mobile, pour tout couple de valeurs δ_1 et δ_2 du paramètre local, l'exécution du programme mobile donne le même résultat observable lorsqu'il appelle le programme local dont le paramètre vaut δ_1 que lorsqu'il appelle le programme local dont le paramètre vaut δ_2.

Si l'on remarque que le programme mobile, en appelant le programme local, peut le placer dans n'importe quel contexte d'utilisation[8], on peut récrire cette propriété ainsi :

> pour tout contexte d'utilisation M, pour tout couple de valeurs initiales δ_1 et δ_2 du paramètre local, l'exécution dans le contexte M du programme local dont le paramètre vaut δ_1 donne le même résultat observable que celle du programme local dont le paramètre vaut δ_2.

Une définition en des termes purement sémantiques se dessine. La sémantique opérationnelle, qui associe à tout programme le résultat de son exécution, induit une équivalence entre programmes, dite contextuelle et définie ainsi :

> deux programmes sont équivalents contextuellement si placés dans n'importe quel contexte d'utilisation, ils donnent le même résultat observable.

Finalement, la propriété de confidentialité s'exprime ainsi :

[8] On néglige le fait que le programme local puisse diverger.

> pour tout couple de valeurs initiales δ_1 et δ_2 du paramètre local, le programme local dont le paramètre vaut δ_1 est équivalent contextuellement au programme local dont le paramètre vaut δ_2.

À ce stade, il est bénéfique de modéliser le raisonnement précédent. Tout programme mobile s'exprime sous la forme d'une fonction qui à un paramètre x représentant l'environnement local associe le programme $\mathsf{M}[x]$, M étant un contexte d'utilisation pour le paramètre x. Si l'on note $\mathsf{L}[\delta]$ le programme local dont le paramètre vaut δ, alors l'exécution du code mobile dans l'environnement local peut être modélisée par l'exécution du programme $\mathsf{M}[\mathsf{L}[\delta]]$. La dernière propriété de confidentialité s'exprime donc sous la forme suivante :

> pour tout couple de valeurs initiales δ_1 et δ_2 du paramètre local, le programme local $\mathsf{L}[\delta_1]$ est équivalent contextuellement au programme local $\mathsf{L}[\delta_2]$.

Cette caractérisation utilisant l'équivalence contextuelle justifie aussi l'utilisation d'une sémantique dénotationnelle, qui associe à chaque programme une dénotation, description abstraite du résultat de son exécution : en effet, une sémantique dénotationnelle, pour être utile, doit au moins être adéquate, ce qui signifie qu'elle doit abstraire correctement, précisément que deux programmes ayant même dénotation sont équivalents contextuellement ; si la réciproque est également vérifiée, la sémantique dénotationnelle est alors complètement adéquate[9]. On peut donc reformuler la propriété de confidentialité en remplaçant l'équivalence contextuelle par l'équivalence dénotationnelle pour obtenir une condition suffisante ou équivalente, suivant que la sémantique dénotationnelle est adéquate ou complètement adéquate, ce qui donne dans notre modèle la condition suivante :

> pour tout couple de valeurs initiales δ_1 et δ_2 du paramètre local, les programmes locaux $\mathsf{L}[\delta_1]$ et $\mathsf{L}[\delta_2]$ ont la même dénotation.

C'est cette voie qui est empruntée par Abadi, Banerjee, Heintze et Riecke dans [2], d'une part, par Sabelfeld et Sands dans [74], d'autre part, bien qu'elle le soit dans un but différent du nôtre : ces deux articles cherchent à décrire les flux d'informations au sein d'un programme donné, sans le placer dans un quelconque contexte, si bien que le rapport entre les équivalences dénotationnelle et contextuelle n'est pas évoqué. Nous retenons cependant que pour exprimer la condition précédente, ces deux articles proposent une interprétation relationnelle s'appuyant sur la sémantique dénotationnelle,

[9] « Fully abstract » en anglais.

qui semble dans ce cas mieux adaptée qu'une sémantique opérationnelle. Précisément, soit Δ la relation binaire entre dénotations définie ainsi :

> le couple (d_1, d_2) appartient à Δ si d_1 et d_2 sont deux dénotations de valeurs possibles pour le paramètre local.

Notons p le paramètre local et $[\![\mathsf{L}[p]]\!]_{(p:d)}$ la dénotation du programme local obtenue en interprétant le paramètre p par une dénotation quelconque d. Il est alors possible de donner une interprétation relationnelle, en liant le paramètre p non plus à une dénotation mais à la relation binaire Δ entre dénotations, et en associant à L la relation notée $[\![\mathsf{L}[p]]\!]_{(p:\Delta)}$ et définie par :

$$[\![\mathsf{L}[p]]\!]_{(p:\Delta)} \stackrel{def}{=} \{([\![\mathsf{L}[p]]\!]_{(p:d_1)}, [\![\mathsf{L}[p]]\!]_{(p:d_2)}) \mid (d_1, d_2) \in \Delta\}\,.$$

La propriété de confidentialité s'exprime finalement ainsi :

> la relation $[\![\mathsf{L}[p]]\!]_{(p:\Delta)}$ est incluse dans la relation diagonale, ensemble des couples (d, d) pour toute dénotation d.

Notre travail a consisté à reprendre ces développements concernant les flux d'informations du point de vue des observations à la frontière. La frontière est ici entendue de manière à la fois globale et permanente, dans le sens où elle est le lieu de l'interaction entre le programme mobile et son environnement local, menant à une observation initiale, caractérisée par la valeur du paramètre local, et une observation finale, le résultat observable de l'exécution du programme mobile dans l'environnement local.

Première question générale et préalable à laquelle nous répondons dans le second chapitre « La sémantique observationnelle » : étant donné un programme, que peut-on en observer ? On considère ici qu'une observation élémentaire consiste à placer le programme dans un contexte d'utilisation et à observer le résultat de son exécution. À la question posée, il est possible de répondre de deux manières, l'une absolue, mettant l'accent sur les observations effectivement associées à chaque programme, l'autre relative, mettant l'accent sur la structure différentielle des observations entre tous les programmes. C'est cette seconde manière qui nous intéresse.
Supposons qu'on ait choisi un ensemble de contextes d'utilisation pour réaliser les observations. Ce choix induit une relation d'équivalence entre programmes, deux programmes étant équivalents si dans tout contexte d'utilisation sélectionné, ils mènent au même résultat observable. La relation d'équivalence la plus fine est évidemment obtenue en choisissant l'ensemble de tous les contextes d'utilisation : on obtient dans ce cas l'équivalence contextuelle induite par la sémantique opérationnelle. Peut-on restreindre l'ensemble des contextes considéré, tout en conservant la même relation d'équivalence induite, soit l'équivalence contextuelle ? Nous répondons par l'affirmative, dans

le cas d'un langage purement fonctionnel, le λ-calcul paresseux avec appel par valeur. Pour observer une fonction, il suffit de l'appliquer à tout argument possible, d'enregistrer l'observation du résultat, la convergence ou la divergence, puis en cas de convergence, de recommencer avec le résultat, qui est une fonction, et ainsi de suite ; autrement dit, on peut se restreindre aux contextes purement applicatifs.
Nous avons mentionné le fait que pour une sémantique dénotationnelle complètement adéquate, l'équivalence dénotationnelle est égale à l'équivalence contextuelle. Nous montrons alors que grâce aux observations ainsi définies, nous pouvons définir une sémantique dénotationnelle complètement adéquate, la sémantique observationnelle : la dénotation d'un programme est simplement son observation.
Comme nous le verrons, l'ensemble de ces résultats est bien connu ; aussi, notre travail consiste surtout en une relecture privilégiant le point de vue observationnel.

Seconde question, relative au contrôle des flux d'informations et traitée dans la première partie du troisième chapitre « Deux analyses de l'environnement local » : comment observer l'environnement local de manière à déterminer s'il vérifie la propriété de confidentialité ? Rappelons que l'environnement local est considéré comme un programme local possédant un paramètre et que la propriété de confidentialité exprime que pour tout programme mobile, les valeurs initiales du paramètre local n'interfèrent pas avec les observations du résultat de l'exécution du programme mobile dans l'environnement local ; plus formellement, si l'on note $\mathsf{L}[\delta]$ le programme local dont le paramètre vaut δ, on a vu que cette propriété se traduit ainsi :

> pour tout couple de valeurs initiales δ_1 et δ_2 du paramètre local, le programme local $\mathsf{L}[\delta_1]$ est équivalent contextuellement au programme local $\mathsf{L}[\delta_2]$,

ou encore en utilisant la sémantique observationnelle,

> pour tout couple de valeurs initiales δ_1 et δ_2 du paramètre local, l'observation du programme local $\mathsf{L}[\delta_1]$ est égale à celle du programme local $\mathsf{L}[\delta_2]$.

Il reste que cette propriété est difficile à manier en raison de la quantification universelle portant sur la relation formée des couples de valeurs possibles pour le paramètre local. Aussi, nous proposons d'interpréter abstraitement la sémantique opérationnelle, le paramètre local étant abstrait par une valeur notée δ représentant la relation formée des couples de valeurs possibles pour le paramètre local. Toujours pour le λ-calcul paresseux avec appel par valeur, nous montrons que si le paramètre local peut être remplacé par n'importe

quel sous-programme, alors la propriété de confidentialité est équivalente à la suivante :

> l'observation abstraite du programme local $\mathsf{L}[\delta]$ ne fait pas apparaître δ.

L'observation abstraite est définie relativement à la sémantique opérationnelle interprétée abstraitement, suivant le même principe générateur que précédemment pour la sémantique observationnelle : ainsi, l'observation d'un programme abstrait est obtenue en l'exécutant abstraitement, en observant son résultat, et s'il converge, en observant son application à tout argument possible. La grande différence relativement à l'observation concrète, c'est que le résultat observable d'une exécution peut être non seulement comme précédemment la convergence ou la divergence, mais aussi une variation, précisément son interprétation abstraite, la valeur δ. Ainsi, si l'observation abstraite du programme local fait apparaître δ, c'est bien que le résultat dépend de la valeur initiale du paramètre local.
Ce critère de confidentialité, obtenu par l'observation abstraite de l'environnement local, induit une méthode permettant d'utiliser pour le code mobile les analyses statiques des flux d'informations, qui lorsqu'elles s'inspirent du travail pionnier de Denning dans [26], s'appuient sur la même interprétation abstraite : à notre connaissance, toutes ces analyses ont été développées pour décrire les flux d'informations au sein d'un programme donné, sans jamais considérer la problématique propre au code mobile, où le programme local, seul analysable, peut être placé dans n'importe quel contexte, mobile.

Contrôler les accès Quelles ressources nécessitent un contrôle d'accès ? Vues du langage de programmation, les ressources correspondent généralement à la source des entrées ou à la destination des sorties, comme le système de fichiers ou le réseau. Ce n'est cependant pas nécessaire, comme le montrent les exemples suivants : on peut souhaiter limiter l'accès à une fonction, c'est-à-dire restreindre ses possibilités d'application, ou encore limiter l'accès au contenu d'une référence, en lecture ou en écriture. De manière générale, les contrôles d'accès concernent donc les opérations appliquées aux valeurs du langage.
En particulier avec Java, sont contrôlés les accès aux méthodes des objets, qu'elles réalisent ou non des entrées ou des sorties. Le langage Java permet de définir une politique de sécurité, sous une forme simple : étant donné un ensemble de sujets[10] et un ensemble de ressources, la politique asso-

[10] Traditionnellement, dans la littérature sécuritaire, on désigne par sujet l'entité active accédant à un objet, entité passive à protéger, qu'on appelle ici ressource.

cie à chaque sujet des permissions d'accès aux ressources. Chaque sujet est chargé d'appliquer la politique aux ressources qu'il contrôle dans son domaine de protection, ensemble de classes qui lui est exclusivement alloué : ainsi, la politique de sécurité est obligatoire, s'imposant à tous les sujets lors de l'exécution, mais sa mise en œuvre discrétionnaire, c'est-à-dire à la discrétion des sujets dans leurs domaines de protection respectifs. Un accès est contrôlé par une vérification préalable de sa permission : le contrôleur d'accès, appelé pour cette vérification, détermine alors si le sujet courant possède cette permission suivant le politique de sécurité, mais aussi si le sujet ayant appelé le sujet courant la possède, et ainsi de suite en remontant la pile des appels. Le sujet courant peut aussi imposer temporairement au contrôleur d'accès de ne pas remonter la pile des appels pour les prochaines vérifications : il prend donc la responsabilité temporaire d'autoriser l'accès indirect aux sujets l'ayant appelé.
Cette technique d'inspection de la pile est implémentée dans Java par instrumentation de la machine virtuelle : voir la description de la méthode `checkPermission` du contrôleur d'accès dans les deux articles des concepteurs de l'architecture de sécurité de Java [30] et [31]. Quant aux contrôles d'accès, ils sont implémentés par une instrumentation du code, qui introduit les appels à la méthode `checkPermission` du contrôleur d'accès. Wallach, dont le travail est à l'origine de l'utilisation de cette technique d'inspection de la pile, propose dans sa thèse [85] une autre implémentation, entièrement par instrumentation du code puisqu'elle repose sur une traduction qui passe effectivement la pile des sujets d'appel en appel[11]. Reprenant l'idée de cette traduction, Pottier, Skalka et Smith développent dans leur article [71] un système de types pour un langage modélisant les aspects sécuritaires de Java[12] : tout programme bien typé s'exécute sans erreur, au sens où chaque contrôle de permission donne toujours un résultat positif ; autrement dit, lors de l'exécution, il est possible de ne pas réaliser les contrôles. Suivant un mouvement bien connu, il serait donc possible de substituer aux contrôles dynamiques de Java, réalisés pendant l'exécution, des contrôles statiques, réalisés avant l'exécution. En réalité, seules les ressources qu'on peut identifier avant l'exécution peuvent être concernées : par exemple, lorsqu'un programme lit le contenu d'un fichier dont le nom ne sera connu qu'à l'exécution (le nom étant calculé, ou entré par l'utilisateur), seul un contrôle dynamique est envisageable.

[11] Pour reprendre les termes de Wallach, c'est une traduction dans le « security-passing style » , néologisme conçu par analogie avec le classique « continuation-passing style ».

[12] Il s'agit du λ-calcul enrichi par des primitives de sécurité, permettant le contrôle des permissions et l'autorisation temporaire d'accès indirect.

En résumé, nous retenons de cette présentation de l'architecture de sécurité de Java le principe suivant : les contrôles d'accès sont réalisés avant l'accès, par instrumentation du code, et permettent de vérifier que l'ensemble des sujets courants obtenu par inspection de la pile possède la permission suivant la politique de sécurité.

Revenons maintenant à notre problématique : comment contrôler les accès du code mobile aux ressources locales ?
Dans l'exemple du langage Java, il suffit d'instrumenter le code local pour réaliser les contrôles d'accès à l'entrée de chaque méthode à protéger. Lors de l'exécution, à chaque entrée dans une méthode instrumentée, le contrôleur d'accès inspecte la pile des appels et donne ou non la permission d'accès suivant la politique de sécurité ; la politique de sécurité, en distinguant deux sujets, correspondant au code mobile et au code local respectivement, permet d'exprimer toutes les variations possibles dans les contrôles. L'efficacité de cette programmation défensive repose sur l'encapsulation : chaque objet, qui encapsule les méthodes permettant d'observer et de modifier son état interne, peut maîtriser les accès à ses méthodes.
Tant qu'il s'agit de contrôler l'accès aux méthodes, il est clair que cette implémentation des contrôles reposant sur l'encapsulation fonctionne parfaitement. Cependant, elle ne permet pas de contrôler l'accès aux références des objets. Bien sûr, si toutes les méthodes d'un objet sont instrumentées, il est inutile de contrôler l'accès à sa référence[13]. Ainsi, le contrôle de l'accès aux références se présente comme une alternative à la programmation défensive utilisant l'encapsulation, alternative qui peut être choisie pour permettre par exemple :

- de réutiliser des classes très utiles, non sécurisées,
- de restreindre les contrôles à l'exécution, dans le but de l'accélérer.

Dans [83], Bokowski et Vitek proposent de contrôler l'accès aux références par confinement[14], voie que nous allons suivre. Pour former une véritable alternative au contrôle d'accès par encapsulation, le confinement doit s'ac-

[13]Généralement, devrait-on ajouter, car on néglige le fait que Java possède certaines opérations applicables aux références, comme le test d'égalité ou la conversion du type, et la possibilité qu'on puisse souhaiter contrôler l'usage de ces opérations pour certaines références.

[14]Le confinement pour les langages orientés objets est le sujet actuellement de nombreux travaux, comme en témoigne le premier atelier « International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO '03) ». Un résultat montré par Banerjee et Naumann dans [10] révèle l'importance de la question. Il s'agit d'une propriété d'indépendance relativement aux implémentations (ou aux représentations des données) confinées : une implémentation confinée peut être changée en une implémentation équivalente sans changer le résultat du programme.

compagner d'un contrôle du code local, permettant de vérifier l'utilisation des références confinées dans l'environnement local.
Pour bien comprendre ces différentes formes de contrôle, un exemple typique, proche de notre exemple liminaire, va nous éclairer ; il nous a été inspiré par un cas réel, à l'origine d'une faille de sécurité, cas relaté dans l'article de Bokowski et Vitek [83, §3].
Considérons un objet r représentant une ressource, muni de deux méthodes, l'une pour modifier son état, `muto`, l'autre pour l'observer, `lego` ; la politique de sécurité stipule que la méthode `muto` de la ressource doit être contrôlée d'une certaine manière, alors qu'aucun contrôle n'est nécessaire pour la méthode `lego`. Supposons que l'environnement doive implémenter une méthode permettant d'observer l'état de la ressource.
Si cette méthode renvoie la référence de r, plus rien n'empêche l'appelant de modifier l'état de r ; c'est là que résidait l'évidente faille de sécurité.
Première solution, par encapsulation : ajouter le contrôle d'accès dans la méthode `muto` de r, sans modifier l'environnement.
Seconde solution, transformer l'environnement en une façade[15] pour la ressource r, composée d'une méthode retournant le résultat de l'appel de la méthode `lego` de r ; les appels de la méthode `muto` de r sont contrôlés dans le code de l'environnement afin de vérifier la politique de sécurité ; si la référence de r est confinée dans l'environnement, aucun autre appel de la méthode `muto` de r ne peut exister dans le code.
Jusqu'à maintenant, nous n'avons pas élucidé le rapport entre les contrôles d'accès dans le code et la politique de sécurité, propriété globale de l'exécution. Dans [41], Jensen et al. proposent ainsi une modélisation d'un programme par un graphe de contrôle ne retenant que les appels de méthodes et les contrôles de permissions, et permettant de simuler abstraitement l'exécution du programme, en particulier la pile des appels ; il est alors possible de vérifier si le graphe de contrôle vérifie une politique de sécurité, exprimée sous la forme d'une propriété portant sur les traces d'exécution. Cette première étude est prolongée par celle de Besson et al. [15], où ce n'est plus un programme entier qui est modélisé, mais une bibliothèque de programmes, afin de permettre une approche modulaire de la vérification. L'analyse du modèle de la bibliothèque produit alors une contrainte portant sur le contexte d'utilisation de la bibliothèque, qu'il suffit de satisfaire pour vérifier la politique de sécurité. Cette dernière approche est adaptée au code mobile, puisque seul l'environnement local peut être modélisé et analysé.

[15] Il s'agit d'un « pattern » de conception bien connu, correspondant à la création d'une interface de haut niveau pour différents services internes.

Comme l'analyse statique déjà vue, la modélisation utilisée dans ces deux derniers articles convient bien pour les contrôles concernant des ressources connues avant l'exécution.
Par la suite, afin d'éviter la question difficile des rapports entre les contrôles dans le code et la politique de sécurité, propriété globale de l'exécution, on ne détaille pas la nature des contrôles réalisés : on suppose seulement que le code est instrumenté de manière à vérifier une certaine politique de sécurité.

Nous nous proposons dans la seconde partie du troisième chapitre « Deux analyses de l'environnement local » d'étudier la question du confinement, question pendante jusqu'à maintenant. Nous l'étudions de deux manières, générale d'une part et appliquée au contrôle d'accès d'autre part, manières que nous présentons maintenant. Nous revenons à notre modèle, l'exécution d'un programme mobile dans un environnement local.
Considérons un ensemble de valeurs du langage, représentant des ressources et devant être confinées au code local. Lors de l'exécution, des opérations peuvent être appliquées à ces valeurs : par exemple, en Java, les valeurs sont des références d'objets, et elles sont utilisées pour appeler les méthodes des objets. On définit le confinement ainsi : une valeur est dite confinée à l'environnement local si lors de l'exécution de tout programme mobile, seules des opérations appartenant au code local lui sont appliquées. Autrement dit, le code mobile n'opère pas directement sur une valeur confinée au code local. Cette définition du confinement nous conduit donc à une nouvelle notion de frontière, locale et dynamique, qui comprend toutes les interactions où le code mobile opère sur des valeurs du code local.
Afin de vérifier la propriété de confinement, nous cherchons idéalement à déterminer les valeurs du code local sur lesquelles le code mobile opère. Pratiquement, nous nous livrons à une approximation, puisque plutôt qu'à la détermination des valeurs, nous nous limitons à celle de leur type. Nous utilisons donc un langage typé, le λ-calcul enrichi de références, permettant la manipulation d'objets en mémoire et entraînant ainsi des effets de bord, ce qui complique les interactions possibles. Nous pouvons étendre aux types la notion de confinement : un type est confiné au code local si toute valeur de ce type est confinée au code local. Inversement, un type est accessible à partir de l'environnement s'il existe un programme mobile pouvant accéder à une valeur du type considéré et d'origine le code local. À partir de l'étude de la frontière entre le code mobile et le code local, nous montrons alors la majoration suivante de l'ensemble des types accessibles à partir de l'environnement :

si un type est accessible à partir de l'environnement, c'est qu'il

> apparaît dans le type de l'environnement en une occurrence positive, ou sous la garde du constructeur des types de références.

Cette majoration, qui ne dépend que du type de l'environnement, est optimale au sens suivant :

> pour tout type A apparaissant dans un type L en une occurrence positive, ou sous la garde du constructeur des types de références, il existe un environnement local de type L tel que le type A est accessible à partir de cet environnement.

Cette caractérisation des types accessibles résout une conjecture de Leroy et Rouaix dans [49, §5.1, p. 397], article dont nous nous inspirons pour l'application du confinement aux contrôles d'accès. Dans cet article, les auteurs proposent (entre autres) de fonder les contrôles d'accès non sur l'encapsulation, mais sur le confinement et le contrôle local des opérations réalisées sur les ressources confinées : cette solution repose donc sur l'abstraction procédurale que constitue l'environnement local.
Le contrôle des opérations s'effectue à leur utilisation ; ainsi, en Java, les contrôles dans le code local seraient placés exactement comme dans l'exemple précédent, soit à l'appel d'une méthode d'un objet dont la référence est confinée, et non dans la méthode appelée. Dans [49, §5.1], pour un langage voisin du nôtre, Leroy et Rouaix proposent un schéma d'instrumentation du code local permettant de contrôler les accès du code local aux ressources à protéger, et montrent que si aucun type d'une ressource à protéger n'apparaît dans le type de l'environnement local, alors le confinement est vérifié, si bien que l'exécution de tout programme mobile dans l'environnement local est sûre. La condition portant sur le type de l'environnement local peut être affaiblie, grâce à notre étude précédente : le confinement est vérifié si aucun type d'une ressource à protéger n'apparaît en mauvaise position dans le type de l'environnement.
Avec cette approche, le contrôle des accès s'appuie donc :

- sur le confinement au code local des ressources à protéger,
- sur l'instrumentation du code, pour contrôler dans le code local les opérations appliquées aux ressources à protéger.

Il nous reste à nous préoccuper de l'instrumentation du code. Plutôt que définir, comme dans l'article de Leroy et Rouaix (cf. [49, §5.1]), un schéma d'instrumentation du code permettant de vérifier une politique de sécurité particulière, nous préférons introduire des opérations instrumentées, en faisant abstraction du détail de leur instrumentation : elles sont donc supposées garantir l'application de la politique de sécurité. Il s'agit de vérifier que l'instrumentation est correctement réalisée au sens suivant : pendant l'exécution

d'un programme instrumenté, seuls les opérateurs instrumentés peuvent accéder aux ressources à protéger. Nous proposons de vérifier statiquement la correction de l'instrumentation par un système de types étiquetés. Sa forme est tout à fait classique, puisque c'est par exemple une restriction du système de types du « SLam Calculus »[16] (cf. [34]), le langage conçu par Heintze et Riecke pour permettre d'exprimer en son sein même une politique de sécurité et de contrôler statiquement son application par son système de types. Pour le confinement, nous utilisons une propriété importante de ce système de types, l'existence d'une relation de sous-typage, qui exprime la possibilité de convertir une valeur qui ne nécessite pas de protection en une valeur à protéger. Nous montrerons que pour confiner, il suffit de convertir, ce qui mettra en évidence la relation entre la conversion et la notion de frontière que nous avons définie et étudiée précédemment.

En résumé, nous proposons deux critères de sécurité pour l'exécution de code mobile ; ils résultent d'analyses de l'environnement local et s'appuient sur deux définitions de la frontière entre le code mobile et l'environnement local,

- la première adaptée au contrôle des flux d'informations entre le code mobile et les ressources à protéger,
- la seconde adaptée au contrôle du confinement des ressources à protéger.

Questions méthodologiques Nous n'avons pas jusqu'à maintenant porté notre attention sur les questions méthodologiques. Les objets manipulés au cours de ce travail ont présenté bien souvent une caractéristique commune, à savoir une structure arborescente éventuellement infinie en profondeur. Citons quelques exemples :

- l'observation d'un programme, telle que nous l'avons définie,
- le calcul infini d'un programme ne terminant pas, tel qu'on peut le décrire avec une sémantique opérationnelle,
- l'état de la mémoire d'un programme en Java, où des objets contiennent des méthodes se référant à d'autres objets, si bien que le référencement (indirect) peut devenir circulaire.

Pour définir ces objets infinis, nous avons privilégié une approche équationnelle ; pour raisonner sur ces objets infinis, nous avons privilégié une approche déductive. Précisément, dans le premier chapitre « Représenter par des arbres », nous étudions successivement

[16] C'est l'acronyme de « Secure λ-Calculus ».

- les systèmes d'équations récursives sur les arbres, afin d'obtenir un moyen simple de définir des arbres éventuellement infinis en profondeur, en particulier des preuves,
- les systèmes d'inférence, en particulier leur interprétation co-inductive, qui autorise des preuves infinies en profondeur, et nous permet de raisonner simplement sur des objets infinis.

L'approche déductive, qui s'appuie sur l'utilisation des systèmes d'inférence et des preuves qu'ils admettent, est peu utilisée, bien qu'elle soit souple et intuitive. Nous essayons de combler cette lacune dans ce chapitre méthodologique, en l'adoptant systématiquement, dans le but de la comparer à l'approche classique par points fixes.

Finalement, la thèse se compose de trois chapitres. Le troisième chapitre expose en détail les deux analyses que nous venons de présenter, concernant respectivement la confidentialité et le confinement. Les deux premiers chapitres sont méthodologiques, et peuvent être lus indépendamment de toute préoccupation sécuritaire ; de ce fait, ils bénéficient chacun d'une longue introduction, exposant en détail leur problématique propre. Il est possible de lire cette thèse en lisant les introductions des chapitres un et deux, puis le chapitre trois, en revenant lorsque c'est nécessaire aux deux premiers chapitres.

Résumons le contenu des trois chapitres :

- le premier chapitre « Représenter par des arbres » présente les techniques utilisées dans les deux chapitres suivants ; s'y ajoutent des résultats et des généralisations permettant de comparer l'approche déductive à l'approche classique par points fixes ;
- le second chapitre « La sémantique observationnelle » concerne l'observation des programmes et son lien avec la sémantique dénotationnelle ; cette étude est menée à partir du λ-calcul paresseux avec appel par valeurs ;
- le troisième chapitre « Deux analyses de l'environnement local » concerne les deux analyses de sécurité, garantissant respectivement la confidentialité et le confinement des ressources ; il est précédé d'une introduction technique, expliquant la méthode commune utilisée pour ces analyses. Noter que pour l'étude du confinement, le λ-calcul est enrichi de références, permettant la manipulation d'objets en mémoire, et devient typé.

La conclusion reprend l'ensemble des résultats, en essayant d'esquisser des extensions possibles. On s'intéresse plus particulièrement à la question de savoir si les techniques retenues pour ce travail conservent leur efficacité

lorsque les langages de programmation utilisés sont étendus de manière à modéliser plus complètement des langages comme Java.

Chapitre 1

Représenter par des arbres

Sommaire

Le chapitre suivant définit différentes sémantiques d'un langage de programmation élémentaire, le λ-calcul paresseux, et utilise largement les méthodes présentées ici. C'est une approche syntaxique qui est suivie pour définir ces sémantiques : partant d'une sémantique opérationnelle, on définit ce que peut être l'observation d'un programme, et on montre que ces observations permettent de définir une sémantique dénotationnelle complètement adéquate, la sémantique observationnelle. Tout au long de ce chapitre, on rencontre des objets infinis, au sens de structures arborescentes non fondées, comme l'observation d'un programme ou une preuve de divergence. Plus simplement, si on considère comme au troisième chapitre la sémantique opérationnelle du λ-calcul enrichi de références permettant la manipulation d'objets en mémoire, on s'aperçoit qu'un état sémantique présente cette

structure arborescente non fondée. Dans cet exemple que nous détaillons dans cette introduction, la sémantique s'exprime sous la forme d'une relation de transition entre des états[1]. Un état sémantique est une abstraction de l'état de la mémoire lors de l'exécution d'un programme (d'où le terme opérationnel). Pour aller vite, un état de la mémoire contient :

- les instructions du programme, parmi lesquelles se distingue l'instruction courante,
- le contenu des objets créés pendant l'exécution.

Quelle est la structure d'un tel état sémantique ?

Prenons l'exemple d'un programme écrit en Java. Une instruction manipule des références (ou adresses) d'objets créés en mémoire pour pouvoir y accéder, et un objet contient des instructions dans ses méthodes. Ainsi, il peut référencer d'autres objets et en particulier s'auto-référencer (directement ou non), si bien qu'on peut considérer que l'état sémantique possède une structure arborescente éventuellement infinie en profondeur, lorsqu'on déplie ces références ; certes, dans ce cas particulier, cette structure peut être représentée de manière finie, grâce à sa régularité.

Comment raisonner sur un tel état sémantique ?

Voyons l'exemple de la sûreté du typage pour des langages typés statiquement, propriété éminemment souhaitable, puisqu'elle affirme que tout programme bien typé ne provoque pas d'erreur de type. Par exemple, pour le langage Java, une erreur de type est la sélection d'un champ ou d'une méthode pour un objet qui en est dépourvu.

Classiquement, pour démontrer la sûreté du typage avec une sémantique opérationnelle, on montre que la propriété de typage se préserve par transition (grâce à un théorème de réduction du sujet[2]) et qu'un état sémantique bien typé ne provoque pas d'erreur. On conclut qu'un programme bien typé ne provoque pas d'erreur par récurrence sur la suite des transitions à partir de l'état initial, formé du programme bien typé. Cette méthode de démonstration est reprise pour le langage fonctionnel et impératif du troisième chapitre (cf. p. 231).

Pour prouver le théorème de réduction du sujet, on définit un système de types pour les états sémantiques, et on essaie de montrer la préservation de la validité d'un jugement de typage dans ce système après réduction du sujet. Si la structure de l'état sémantique n'est pas fondée, il est impossible de construire une preuve de typage dans ce système par récurrence structurelle

[1] On considère donc une sémantique opérationnelle « small step » (à petit pas, suivant l'exécution), et non « big step » (à grand pas, donnant directement le résultat).

[2] Cette terminologie est relative au jugement de typage, où le sujet est l'état sémantique, l'objet étant le type.

sur l'état. Deux solutions ont été apportées à ce problème :

1. une représentation bien fondée de l'état sémantique est utilisée, et on peut dans ce cas recourir à la récurrence structurelle,
2. on interprète de manière co-inductive le système de types.

L'interprétation co-inductive consiste à définir l'ensemble des jugements de typage valides comme étant le plus *grand* point fixe de l'opérateur d'inférence associé au système de types, et non le plus *petit* point fixe. Dans ce cas, pour montrer que des jugements de typage sont valides, il suffit de montrer la densité de leur ensemble. L'approche co-inductive, permettant d'éviter de recourir à une quelconque représentation, est plus directe, mais moins intuitive, ne s'appuyant pas sur des preuves. Par la suite, nous essayons de réunir le meilleur des deux approches : l'utilisation de preuves, proche de l'intuition, et l'interprétation co-inductive, généralisant l'inductive au cas des objets infinis. La solution est simple :

> les objets infinis nécessitent des preuves infinies.

Si nous avons traité l'exemple de la sûreté du typage, c'est par fidélité historique, dans la mesure où c'est dans une telle démonstration que le terme de co-induction a été utilisé pour la première fois en informatique, et où les démonstrations de ce résultat utilisent l'une ou l'autre des techniques précédentes, lorsque les langages sont suffisamment expressifs. De plus, les travaux sur cette question sont nombreux. C'est qu'en effet le typage statique a pour but fondamental d'éviter toute erreur de type à l'exécution : telle est en tout cas l'opinion communément admise, comme en témoigne l'article didactique « Type Systems » de Cardelli [18]. Pour un langage typé statiquement, la sûreté du typage garantit que ce but est atteint. Pour des langages de bas niveau, privilégiant l'efficacité, au détriment de l'abstraction, comme le langage C par exemple, le typage statique n'est pas sûr. Pour des langages de haut niveau comme Java ou ML, qui manipulent des entités abstraites, la sûreté du typage est acquise.
Pour Java, cette propriété a été recherchée dès la conception[3], et elle a été par la suite montrée, par exemple par Drossopoulou et al. ([27]). Dans cet article, c'est la représentation finie de l'état sémantique qui est utilisée.
Pour ML, la situation s'avère plus intéressante d'un point de vue méthodologique, car de nombreux travaux ont été consacrés à la sûreté de son système de types, qui autorise le polymorphisme. On pourra consulter l'introduction de l'article de Wright et Felleisen [86] pour une revue complète

[3]On trouve dans la spécification la description informelle des règles de typage bien connues.

des méthodes utilisées. Si l'on s'intéresse aux principaux travaux concernant le noyau fonctionnel augmenté de références, alors, conformément à notre description précédente, il est effectivement possible de les répartir en deux groupes :

- d'un côté, l'étude initiale de Milner et Tofte[4], suivie par celle de Talpin et Jouvelot [80], où la co-induction est utilisée,
- de l'autre, toutes les autres études (par exemple celles de Leroy [46] et Harper [33], qui mentionnent explicitement l'avantage d'une représentation finie sur la co-induction).

Noter que la démonstration que nous donnons au troisième chapitre tombe aussi dans cette seconde catégorie, ce qui confirme cette tendance.

Revenons aux objets infinis rencontrés dans notre exemple : ils correspondent ou bien à des termes, au sens large (les états sémantiques), ou bien à des preuves. Dans les deux cas, ils peuvent être représentés concrètement par des arbres, dont des branches peuvent être infinies. Explorer les possibilités de représentation par arbres, telle est l'idée centrale de cette partie méthodologique. En premier lieu, on va déterminer un moyen pour définir des arbres : ce sera par la résolution de systèmes d'équations récursives. En second lieu, on montrera que raisonner, c'est prouver : l'induction et la co-induction, tant dans un système d'inférence que dans un treillis complet, seront caractérisées en utilisant des arbres de preuves, plutôt que des points fixes.

Équations récursives et arbres Comment définir des arbres, éventuellement infinis en profondeur ? Nous répondons : par des systèmes d'équations récursives.

Par exemple, considérons la signature formée de deux symboles, 0 et S, d'arité zéro et un respectivement, représentant la valeur nulle et la fonction successeur. Elle permet d'engendrer librement l'ensemble des entiers naturels, qu'on peut représenter sous la forme d'arbres, ou mieux, en notation préfixe, de mots, de la manière suivante : 0, $S\,0$, $S\,S\,0$, etc. Le premier ordinal limite, ω, peut être représenté par un mot infini, noté $S \ldots S \ldots$, ou S^ω, en notation préfixe, et ce mot est l'unique solution de l'équation $x = S\,x$.

[4]Dans sa thèse [81], Tofte attribue à Milner l'idée d'utiliser la co-induction, principe présenté par ailleurs dans un article commun [56]. Dans ce dernier, le terme de « co-induction » est forgé pour la première fois pour indiquer la méthode duale de l'induction ; ce sont les fermetures des fonctions récursives qui justifient l'usage de la co-induction, jouant le même rôle que les références dans notre exemple. L'idée de la co-induction remonte à des travaux précédents de Milner, concernant la bisimilarité de processus concurrents, comme nous allons le voir plus loin.

Évidemment, une équation de la forme $x = x$ n'admet pas une unique solution, mais une infinité. Ainsi, seules les équations gardées sont considérées, soit des équations de la forme

$$x = f(\ldots)\,,$$

où le membre droit $f(\ldots)$ est un arbre gardé par un symbole fonctionnel, f, et non par une inconnue. Une solution est une valuation, application associant à toute variable inconnue un arbre, qui vérifie les équations après qu'on a substitué aux inconnues les arbres associés par la valuation. On montrera par la suite l'existence et l'unicité de la solution de systèmes d'équations récursives gardées. Nous traitons deux cas, qui se distinguent par les arbres considérés, d'une part, les arbres étiquetés, sans aucune contrainte de formation, d'autre part, les arbres respectant une signature à plusieurs sortes (des termes, donc).
Par souci de simplification, nous avons cherché une formulation unique de ces deux cas, en décrivant les arbres étiquetés comme des arbres respectant une certaine signature, ce qui nous a imposé d'étendre la définition habituelle des signatures : en effet, les arbres étiquetés peuvent être de degré infini, ce qui n'est pas le cas d'un terme respectant une signature classique. Notre définition des signatures est directement inspirée de l'approche catégorique de l'algèbre universelle, où les signatures sont remplacées et généralisées par des foncteurs. Reprenons l'exemple de la signature formée de deux symboles, 0 et S, d'arité zéro et un respectivement, représentant la valeur nulle et la fonction successeur. À cette signature, dans la catégorie des ensembles, on peut associer le foncteur F défini par $F(X) = 1 + X$, où 1 est un ensemble singleton. Une algèbre sur la précédente signature devient une algèbre du foncteur F, soit un couple formé d'un ensemble A et d'une application ι de $F(A)$ dans A : les restrictions de ι à 1 et A donnent l'interprétation respectivement de 0 et S. Généralement, pour une signature à une seule sorte, le foncteur F aura donc une forme polynomiale :

$$F(X) = \Sigma_f X^{j_f}\,,$$

où f parcourt l'ensemble des opérations de la signature et j_f représente l'arité de f. Notre généralisation consiste à considérer des formes polynomiales plus générales, puisqu'admettant des produits cartésiens quelconques :

$$F(X) = \Sigma_f X^{J_f}\,,$$

où pour chaque opération f, J_f est cette fois-ci un ensemble, et non un entier.

Les résultats élémentaires évoqués concernant la résolution des équations récursives sont évidemment bien connus, mais sont néanmoins le point de départ de développements conséquents. Ainsi, les systèmes d'équations portant sur les arbres infinis ont été particulièrement étudiés par Courcelle (cf. [21] par exemple), et c'est la théorie des schémas de programmes[5] qui réclamait une telle étude préliminaire.
Le terme de garde est surtout utilisé dans la description des processus concurrents par des équations récursives, de manière analogue à notre propre utilisation (cf. par exemple l'ouvrage de Baeten et Weijland [9, §2.3]) ; de même, dans la théorie des types, considérée comme une théorie de définitions inductives et co-inductives, les termes infinis des types co-inductifs sont introduits par des équations récursives gardées, les types co-inductifs servant généralement à décrire des processus (cf. l'article de Coquand [20], pour la première utilisation de cette notion en théorie des types).

Les systèmes d'équations récursives ne sont pas faciles à manier en l'état. En particulier, ils ne permettent pas de prendre en compte facilement l'action d'opérations définies sur les arbres. Par exemple, considérons un système de la forme

$$\begin{array}{lcl} x & = & f(\dots y \dots)\,, \\ y & = & a_y\,, \\ \vdots & \vdots & \vdots\,. \end{array}$$

Supposons qu'on s'aperçoive que la valuation recherchée ne vérifie pas la première équation, mais une équation voisine, de la forme

$$x = f(\dots \alpha(y) \dots)\,,$$

où les occurrences de y ont été transformées en l'application à y de l'opération unaire α définie sur l'ensemble des arbres. Pour retrouver un système auquel les résultats précédents d'existence et d'unicité s'appliquent, on est obligé de modifier l'équation $(y = a_y)$, pour y reporter l'action de α, ce qui peut entraîner de nouvelles modifications pour les équations des inconnues apparaissant dans a_y. Évidemment, pour que ce procédé soit correct, α doit vérifier certaines conditions. On cherche donc à montrer un théorème d'existence et d'unicité de la solution pour des systèmes où des opérations entre arbres interviennent dans les équations, théorème valable pour un ensemble d'opérations vérifiant certaines conditions.

[5] Le lecteur intéressé par ce sujet pourra consulter l'article synthétique de Courcelle [23], et en particulier les notes historiques concluant l'article (p. 488).

Une première solution consiste à considérer les opérations comme des étiquettes, en l'occurrence des symboles fonctionnels, et à réaliser le quotient de l'ensemble des arbres par une congruence, de telle manière que chaque classe d'équivalence contienne une unique forme normale, un arbre sans opération, représentant le résultat du calcul de chaque arbre de la classe. C'est la voie empruntée par Courcelle, dans [22][6] : la congruence, définie initialement comme la plus petite congruence vérifiant certaines équations sur les arbres finis, est étendue aux arbres infinis, en calculant la forme normale d'un arbre infini comme la borne supérieure des formes normales des arbres finis l'approchant, l'approximation se définissant à partir de la relation d'ordre « est moins défini que » (la définition de cette relation se trouve dans le paragraphe suivant). Dans ce cas, la condition de garde n'est pas suffisante pour assurer l'unicité de la solution. Par exemple, considérons la signature contenant en plus de la valeur de non-définition $\perp$, un symbole d'arité un, le constructeur c, et adjoignons à cette signature l'opération unaire qu'on souhaite utiliser dans les équations, soit le destructeur d. Définissons la congruence à partir des équations $d \perp = \perp$ et $d(c(e)) = e$, pour tout terme fini e ; les formes normales finies sont de la forme $c^n \perp$, la seule forme normale infinie étant c^ω. Clairement, l'équation $x = d(c(x))$ admet une infinité de solutions, bien qu'elle soit gardée.
Comme notre intention est plutôt de conserver des équations gardées, et de qualifier les opérations autorisées dans de telles équations, ce n'est pas cette solution que nous avons adoptée. Notre solution est issue d'un transfert : elle provient en effet du développement d'une théorie des ensembles, sans l'axiome de fondation, remplacé par un axiome, dit d'anti-fondation, affirmant l'existence et l'unicité de la solution de certains systèmes d'équations récursives sur les ensembles : c'est cette possibilité de résoudre des équations récursives qui est à l'origine de l'intérêt pour cette théorie, présentée en détail dans l'ouvrage d'Aczel [7]. Les équations considérées sont de la forme $x = \{\ldots y \ldots\}$; si on représente les ensembles par leur structure arborescente, définie à partir de la relation d'appartenance, l'analogie avec les équations gardées sur les arbres devient claire. Ainsi, l'équation précédente se récrit sous la forme $x =\ni (\ldots y \ldots)$: l'étiquette de garde $\ni$ est l'étiquette utilisée pour les ensembles non vides, les sous-arbres sous cette étiquette représentant les éléments de l'ensemble. Dans l'ouvrage de Barwise et Moss [13], concernant des applications de cette théorie des ensembles, la même préoccupation que la nôtre se retrouve : autoriser des opérations dans les

[6]Les cinq premières pages décrivent précisément le contenu de l'article, et pourront compléter utilement notre brève présentation.

équations (cf. [13, p. 239]). Nous avons donc adapté et généralisé la solution qui y est proposée.
Le membre droit d'une équation peut désormais comporter des opérations autorisées. Cependant, afin de pouvoir définir facilement la solution d'un système, il est nécessaire d'imposer une condition portant sur l'utilisation des opérations autorisées. Barwise et Moss dans [13, p. 239] imposent d'utiliser les opérations en les appliquant uniquement à des variables. Nous relâchons cette contrainte en permettant de construire inductivement les membres droits des équations à partir de ce cas de base, suivant les deux règles de construction suivante :

- si $(a_i)_i$ est une famille de membres droits, alors pour toute étiquette f, $f(a_i)_i$ est aussi un membre droit,
- si $(a_i)_i$ est une famille de membres droits, alors pour toute opération autorisée α, $\alpha(a_i)_i$ est aussi un membre droit,

Dans ce cas, une solution d'un tel système est une valuation vérifiant les équations après qu'on a remplacé les inconnues par les arbres associés par la valuation et effectué l'évaluation des opérations. Nous nous intéressons alors à la résolution d'un système d'équations gardées, c'est-à-dire dans lesquelles toute inconnue est gardée par au moins une étiquette : nous cherchons à déterminer des conditions portant sur les opérations autorisées dans les équations, suffisantes pour assurer l'existence et l'unicité d'une solution. Nous verrons que si les opérations autorisées peuvent s'étendre à des arbres contenant des variables et des opérations appliquées à des variables (comme dans le cas de base) et en préserver la propriété de garde, alors on peut se ramener à la résolution d'un système classique d'équations gardées, à condition que les opérations autorisées commutent avec les valuations. Telle est donc la propriété principale que doit vérifier l'ensemble des opérations autorisées : c'est également la propriété donnée par Barwise et Moss dans [13, Th. 16.11]. Un exemple intéressant d'opérations autorisées, c'est l'ensemble des applications sur les étiquettes, prolongées de manière naturelle sur les arbres.

Pour résumer, les systèmes d'équations récursives permettent de définir des objets infinis sous la forme d'arbres ; dans les équations, afin de faciliter l'utilisation de cette méthode de définition, certaines opérations sont autorisées.

Raisonner sur les objets infinis Prenons l'exemple mentionné précédemment de la relation d'ordre « est moins défini que », définie sur l'ensemble des arbres (augmenté de l'arbre vide, noté $\perp$) et notée $\leq$. Dans ce

cas, la relation d'ordre est définie par un système d'inférence, formé des règles[7] suivantes :

$$\frac{\emptyset}{(\bot, a)} \quad (a \text{ arbre}),$$

$$\frac{(a_i, b_i)_{i\in I}}{(f(a_i)_{i\in I}, f(b_i)_{i\in I})} \quad \left(\begin{array}{l} f \text{ étiquette} \\ (a_i)_i, (b_i)_i : \text{sous arbres} \end{array} \right).$$

La première règle, qui est un axiome puisqu'il n'y a pas de prémisses, indique que l'arbre vide est le plus petit élément ; la seconde permet de déduire, pour toute étiquette f, $f(a_i)_{i\in I} \leq f(b_i)_{i\in I}$ des inégalités $(a_i \leq b_i)_{i\in I}$: elle s'interprète donc comme la croissance de f. Par définition, un jugement $a \leq b$ est vrai s'il peut être prouvé dans le système précédent : cela signifie qu'ou bien $a = \bot$, ou bien a et b commencent par la même étiquette, et les sous-arbres respectifs de a et b sont correctement ordonnés. Si a et b ne sont pas fondés, la preuve ne termine donc pas : elle est infinie en profondeur.
Généralisons : on cherche à prouver des jugements portant sur des objets infinis (ou sur des n-uplets d'objets), au sein d'un système d'inférence. La preuve de la validité d'un jugement sur un objet se fait en combinant des règles d'inférence du système : un jugement se déduit d'autres jugements, si ceux-ci forment dans une règle ses prémisses, jugements qui eux-mêmes se déduisent, et ainsi de suite. Une preuve forme bien alors un arbre, où chaque nœud contient un jugement et a pour sous-arbres les preuves de ses prémisses suivant une règle du système d'inférence ; la conclusion de la preuve est alors à la racine de l'arbre. Une telle preuve peut être bien fondée ou non : dans ce dernier cas, on peut construire en remontant la preuve à partir de sa conclusion au moins une suite infinie de déductions.
Comme les preuves peuvent être représentées par des arbres, il est possible de les définir en utilisant des systèmes d'équations récursives. Reprenons l'exemple des entiers naturels, représentés par les arbres respectant la signature formée des deux symboles 0 et S, d'arité zéro et un respectivement. Pour montrer que $S^\omega \leq S^\omega$, on utilise la seule règle applicable

$$\frac{(S^\omega, S^\omega)}{(S\,S^\omega, S\,S^\omega)}.$$

[7] Il s'agit plutôt d'un schéma de règles, qui définit un ensemble de règles, obtenu en parcourant toutes les valeurs possibles des paramètres libres du schéma. On suivra toujours cette convention : une règle possédant des paramètres libres est un schéma de règles, les paramètres libres étant quantifiés universellement.

La preuve est infinie puisque la prémisse est identique à la conclusion, et peut être vue comme l'unique solution du système suivant :

$$x \quad = \quad \frac{x}{(S^\omega, S^\omega)}.$$

Nous allons caractériser la validité des jugements dans un système d'inférence en admettant certaines preuves et en refusant les autres ; ainsi, une interprétation d'un système d'inférence dégage un ensemble de preuves, dites admissibles, et un ensemble de jugements, dits valides, qui sont les conclusions des preuves admissibles : on dira que ces conclusions sont validées par leurs preuves, admissibles. L'interprétation classique utilise les preuves bien fondées, une autre, plus originale, accepte aussi les preuves infinies en profondeur. Ainsi, dans l'article d'Aczel [6], souvent cité, qui présente les définitions inductives et co-inductives (sous le nom de noyau) au sein d'un système d'inférence, seules les preuves bien fondées sont utilisées pour caractériser les définitions inductives. C'est dans l'article de Coquand [20], concernant la théorie des types et déjà cité, qu'est introduite la notion de preuves définies récursivement, et donc potentiellement infinies en profondeur : par l'isomorphisme de Curry-Howard, un terme s'interprète en effet comme une preuve, et nous avons vu précédemment qu'un terme d'un type co-inductif peut être défini par un système d'équations récursives gardées.
Nous allons distinguer deux ensembles de preuves admissibles :

- l'ensemble des preuves *bien fondées*, pour lesquelles tout chemin à partir de la conclusion est fini,
- l'ensemble de *toutes* les preuves, où les chemins peuvent être finis comme infinis.

Ces deux interprétations, inductives et co-inductives respectivement, permettent en effet de caractériser deux ensembles de jugements remarquables, les plus petit et plus grand points fixes de l'opérateur d'inférence associé au système : c'est cette multiplicité de points de vue qui nous a paru particulièrement intéressante à étudier, avec d'une part, l'approche déductive, utilisant les preuves, d'autre part, l'approche algébrique, s'appuyant sur les points fixes. Avant de développer cette comparaison, d'autres ensembles de preuves méritent cependant d'être mentionnés, comme nous allons le voir maintenant.

Digression : d'autres ensembles de preuves remarquables Voyons tout d'abord un ensemble intermédiaire entre celui des preuves bien fondées et celui de toutes les preuves. Une preuve est *régulière* si sur tout chemin infini partant de la racine, on ne rencontre qu'un nombre fini de sous-preuves.

Toute preuve bien fondée est ainsi régulière. Quant à une preuve régulière, bien qu'elle puisse être infinie en profondeur, elle possède un caractère bien fondé, moyennant une représentation particulière, que nous détaillons. Associons à tout système d'inférence Φ un nouveau système, Φ', tel que toute preuve régulière dans Φ corresponde à une preuve bien fondée dans Φ'. Les jugements de Φ' sont des *séquents* qui, comme en déduction naturelle, ont pour antécédent une suite de jugements de Φ, et pour thèse un jugement de Φ ; on note $J \vdash j$ un tel séquent, où J est une suite de jugements et j un jugement. Les règles de Φ' se construisent à partir de celles de Φ ainsi. Considérons une règle (Z, z) de Φ, ce qui signifie que de l'ensemble de prémisses Z, on déduit la conclusion z. On lui associe les règles $(\{(J\, z \vdash y) \mid y \in Z\}, (J \vdash z))$, pour toute suite de jugements J ne contenant pas z. L'idée est ainsi de considérer que la thèse d'un séquent peut servir d'antécédent dans une prémisse : pour en tirer parti, on ajoute donc l'axiome classique $(\emptyset, (J \vdash z))$ si z appartient à J. C'est cet axiome qui permet d'arrêter la traduction dans Φ' de toute preuve régulière dans Φ, permettant ainsi d'obtenir une preuve bien fondée. Précisément, un jugement j est validé par une preuve régulière dans Φ si et seulement si le séquent $\emptyset \vdash j$ est validé par une preuve bien fondée dans Φ'.
Reprenons l'exemple précédent, où il s'agit de montrer que $S^\omega \leq S^\omega$. La règle

$$\frac{(S^\omega, S^\omega)}{(S^\omega, S^\omega)}$$

donne dans le système associé les règles

$$\frac{J\,(S^\omega, S^\omega) \vdash (S^\omega, S^\omega)}{J \vdash (S^\omega, S^\omega)},$$

si bien que la preuve régulière infinie en profondeur

$$\cfrac{\cfrac{\cfrac{\cdots}{(S^\omega, S^\omega)}}{(S^\omega, S^\omega)}}{(S^\omega, S^\omega)}$$

devient une preuve bien fondée, soit

$$\cfrac{\cfrac{\emptyset}{(S^\omega, S^\omega) \vdash (S^\omega, S^\omega)}}{\emptyset \vdash (S^\omega, S^\omega)} \quad .$$

À notre connaissance, cette transformation a été uniquement étudiée dans le cas où les preuves infinies sont nécessairement régulières, du fait de la structure régulière des jugements. On peut par exemple citer l'étude de Brandt et Henglein [17], concernant les relations d'égalité structurelle et de sous-typage pour les types récursifs, où la transformation repose sur la même idée d'utiliser des séquents.
Pour terminer cet exemple, mentionnons le fait que l'ensemble des conclusions des preuves régulières forme un point fixe de l'opérateur d'inférence.

D'autres ensembles de preuves admissibles sont envisageables. Par exemple, on peut envisager de limiter l'application d'une règle d'inférence suivant son contexte d'utilisation dans une preuve ; dans l'article des Cousot [24], certainement à l'origine de cette idée, c'est la profondeur des preuves des prémisses qui est prise en compte. Cette méthode est utilisée dans les chapitres suivants à plusieurs reprises (cf. p. 106 pour une introduction, p. 116 et suivantes pour un traitement détaillé, p. 196 et p. 202 pour deux exemples). Dans ce chapitre, nous donnons seulement une définition d'une telle interprétation, dite mixte, ainsi qu'une manière de construire des preuves admissibles. Si nous n'avons pas étudié plus avant cette interprétation, c'est qu'elle ne mène pas à d'autres caractérisations intéressantes ; par exemple, la caractérisation comme point fixe d'un opérateur sur les jugements est peu intuitive, contrairement à celle par les preuves : c'est en tout cas la conclusion que nous tirons de la lecture de l'article [24, §1.4], où les auteurs Cousot et Cousod sont cependant moins catégoriques (« may be less intuitive », écrivent-ils).
En conclusion, cette possibilité de varier l'ensemble des preuves admissibles nous semble un avantage décisif de l'approche déductive, lui apportant une grande souplesse.

Par la suite, nous allons comparer pour un système d'inférence l'approche déductive, utilisant les preuves, à une autre approche classique, par points fixes. Nous nous limitons à l'ensemble des preuves bien fondées et à celui de toutes les preuves.

Caractérisation des points fixes À tout système d'inférence, il est possible d'associer un opérateur croissant sur les jugements, donnant pour tout ensemble de jugements les conclusions qu'on peut en déduire en une inférence : c'est l'opérateur d'inférence. Il admet un *plus petit point fixe* et un *plus grand*, qui peuvent être caractérisés[8]

[8]Nous nous appuyons sur l'article de Kanamori [43] pour les références historiques qui suivent et que nous ne reprenons pas dans la bibliographie ; elles concernent des résultats

– de manière imprédicative[9], comme plus *petite* partie *stable* et plus *grande* partie *dense* (dual de stable) respectivement, grâce au théorème de Tarski[10],
– de manière itérative, par induction transfinie, cas particulier du résultat valable pour toute application croissante sur un ensemble ordonné complet[11] (corollaire du théorème de Bourbaki (1939), où la référence aux ordinaux est remplacée par l'affirmation de l'existence d'un ensemble bien ordonné).

La caractérisation par les preuves que nous allons donner est à la fois prédicative et libre de tout recours aux ordinaux[12]. Nous la qualifions de déductive, ce qu'elle est essentiellement, puisqu'elle s'exprime ainsi :

– un jugement appartient au *plus petit point fixe* de l'opérateur d'inférence si et seulement s'il est validé par une preuve *bien fondée*,
– un jugement appartient au *plus grand point fixe* de l'opérateur d'inférence si et seulement s'il est validé par une preuve *quelconque*.

Ainsi se justifient que le plus petit point fixe soit appelé l'ensemble engendré *inductivement* par le système d'inférence, de même que le plus grand soit appelé l'ensemble engendré *co-inductivement* : il suffit de penser à la caractérisation correspondante par les preuves.

Les caractérisations imprédicatives et itératives concernent toute application croissante définie sur un treillis complet, et pas seulement un opérateur croissant sur un ensemble. Il est donc naturel de se demander s'il est encore possible de donner une caractérisation déductive dans le cas général d'un treillis complet. Les données du problème sont donc modifiées ainsi : l'ensemble des jugements est supposé former un treillis complet, et on considère une application croissante définie sur les jugements, η. La relation d'ordre entre jugements peut être interprétée comme une règle d'affaiblissement : si z est inférieur à y et si y a été prouvé, alors on peut déduire z. Une interpré-

mathématiques classiques, développés initialement en théorie des ensembles, considérée comme la théorie unifiant les mathématiques, et finalement utilisés pour fonder la sémantique dénotationnelle des langages de programmation.

[9] Dans le sens élémentaire où chaque point fixe appartient à l'ensemble à partir duquel il est défini.

[10] Ou de Knaster-Tarski, Knaster l'ayant montré en 1928 dans le seul cas du treillis complet formé des parties d'un ensemble, Tarski le généralisant à tout treillis complet (en 1939, repris dans son étude classique de 1955).

[11] Un ensemble ordonné est *complet* si toute chaîne y admet une borne supérieure.

[12] En ce sens, elle respecte l'intention de Kuratowski, qui dans son article de 1922, cherche « à éliminer les nombres transfinis des raisonnements mathématiques », le théorème du point fixe de Kuratowski étant généralisé par celui de Bourbaki. Cependant, s'exprimer en terme de bon ordre ou d'ordinal est évidemment équivalent.

tation en termes d'information se dessine : l'inégalité $z < y$ signifie alors que z contient moins d'information que y. Ainsi, le plus petit élément du treillis peut être interprété comme l'absence d'information, et plus généralement, l'information contenue dans une partie du treillis correspond à l'information contenue dans la borne supérieure de cette partie. L'interprétation de η par un système d'inférence devient claire : le système contient, en plus des règles d'affaiblissement, les règles $(Z, \eta(\vee Z))$, pour toute partie Z. On montre alors les correspondances suivantes :

- l'ensemble engendré *inductivement* par le système d'inférence associé est égal à l'idéal engendré[13] par le *plus petit point fixe* de η,
- l'ensemble engendré *co-inductivement* est égal à l'idéal engendré par le *plus grand point fixe* de η.

Cette démonstration est menée en formalisant l'interprétation précédente en termes d'information, par la représentation classique du treillis complet dans l'ensemble de ses parties, qui utilise les idéaux principaux (c'est-à-dire les idéaux engendrés par les éléments du treillis).
La définition par les preuves bien fondées de l'ensemble engendré inductivement par un système d'inférence est bien connue : on la trouve par exemple dans l'introduction d'Aczel aux définitions inductives [6]. Pour l'ensemble engendré co-inductivement, cette définition par les preuves est rarement utilisée, pas par exemple dans l'article précédent d'Aczel. Dans l'article de Coquand [20], qui concerne les objets infinis en théorie des types, et où des preuves infinies sont utilisées, c'est sous une forme légèrement différente que l'ensemble engendré co-inductivement est caractérisé (cf. [20, §2.4]) : elle est néanmoins équivalente à la caractérisation par les preuves lorsque le système d'inférence est déterministe[14]. En ce qui concerne les treillis complets, cette approche déductive n'est pas utilisée, à notre connaissance.

Utiliser les preuves pour raisonner Les preuves peuvent être utilisées directement pour valider des jugements. On peut les construire à partir de systèmes d'équations récursives ; cette méthode est particulièrement adaptée lorsque c'est l'ensemble engendré co-inductivement qui est considéré, alors que pour l'ensemble engendré inductivement, il est nécessaire en plus de vérifier la bonne fondation des preuves ainsi construites. Dans ce dernier cas, on préfère donc partir de la bonne fondation des preuves pour raisonner par

[13] L'idéal engendré par un élément z est la section initiale fermée engendrée par z, soit l'ensemble des y inférieurs ou égaux à z.

[14] Un système d'inférence est *déterministe* si deux règles ayant même conclusion sont égales : la conclusion d'une règle détermine ainsi ses prémisses, ce qui implique qu'un jugement est validé par une unique preuve.

récurrence structurelle. Ainsi se dessinent deux *principes*, l'un *co-inductif*, l'autre *inductif* ; étant donné un système d'inférence Φ et une partie Z de l'ensemble des jugements, ils permettent de montrer respectivement que

- Z est inclus dans l'ensemble engendré co-inductivement par Φ, si l'on peut construire des preuves validant les jugements de Z,
- l'ensemble engendré inductivement par Φ est inclus dans Z, si l'on peut montrer l'appartenance à Z par récurrence structurelle sur les preuves bien fondées.

Ces principes sont optimaux dans le sens où la conclusion qu'ils permettent de déduire est équivalente à la condition à vérifier pour appliquer le principe. À fin de comparaison, il est utile de les décliner suivant les deux autres caractérisations données, imprédicatives et itératives. Considérons l'opérateur d'inférence φ associé à Φ. Il est possible de calculer le plus petit point fixe à partir de l'ensemble vide, puis en itérant par φ, en prenant pour le premier ordinal limite, ω, la réunion des itérés, et en continuant ainsi jusqu'à obtenir une stabilisation. Soient $(\Delta_\alpha(\varphi))_{\alpha \text{ ordinal}}$ cette suite d'itérés, stationnaire à partir d'un certain rang, et $(\nabla_\alpha(\varphi))_{\alpha \text{ ordinal}}$ celle obtenue par dualité et permettant de calculer le plus grand point fixe. Pour les deux caractérisations, voici les conditions permettant de déduire pour le principe inductif que le plus petit point fixe de φ est inclus dans Z, pour le principe co-inductif que Z est inclus dans le plus grand point fixe de φ :

caractérisation \ principe	inductif	co-inductif
imprédicative	stabilité de $Z \cap \mathrm{lfp}(\varphi)$ $\varphi(Z \cap \mathrm{lfp}(\varphi)) \subseteq Z$	densité de $Z \cup \mathrm{gfp}(\varphi)$ $Z \subseteq \varphi(Z \cup \mathrm{gfp}(\varphi))$
itérative	induction transfinie	
	$\forall\, \alpha\,.\ (\ \forall\, \beta < \alpha\,.\ \Delta_\beta(\varphi) \subseteq Z\ ) \Rightarrow \Delta_\alpha(\varphi) \subseteq Z$	$\forall\, \alpha\,.\ (\ \forall\, \beta < \alpha\,.\ Z \subseteq \nabla_\beta(\varphi)\ ) \Rightarrow Z \subseteq \nabla_\alpha(\varphi)$

Avec ces conditions, tous les principes sont optimaux ; autrement dit, pour chaque principe, toutes ces conditions sont équivalentes entre elles et à la conclusion qu'elles permettent de déduire. La comparaison qui suit n'est donc pas logique, mais méthodologique. On cherche à interpréter en termes de preuves les conditions tirées des caractérisations imprédicatives et itératives.

Pour le principe inductif, lorsqu'on raisonne par récurrence structurelle sur les preuves, on montre, règle par règle, la condition de stabilité. Cette condition de stabilité est généralement donnée sous une forme plus forte, puis-

qu'elle concerne la stabilité de Z et non celle de $Z \cap \mathrm{lfp}(\varphi)$: le principe associé, qui n'est pas optimal, est souvent appelé le principe inductif de Park, en référence à son article précurseur [62]. La forme plus faible, utilisée ici, qui donne un principe optimal, est très courante : en témoigne par exemple son utilisation dans des démonstrateurs de théorèmes, comme Isabelle [64, §2.3].
Quant à l'induction transfinie, elle correspond à l'ordonnancement des preuves suivant leur hauteur, plus exactement suivant une notion étendue de la hauteur, exprimée par un ordinal. La hauteur d'un arbre fini est donnée par un entier naturel, mesurant la longueur du plus grand chemin de la racine à une feuille, et est définie par l'équation récursive suivante :

$$h(f(a_i)_{i\in I}) = \bigvee_{i\in I} h(a_i) + 1\,,$$

où $f(a_i)_{i\in I}$ est l'arbre étiqueté par f à la racine et ayant pour sous-arbres immédiats les arbres de la famille $(a_i)_{i\in I}$. Cette définition peut être étendue aux arbres bien fondés, à condition de l'interpréter dans la classe des ordinaux, et non seulement dans l'ensemble des entiers. C'est qu'en effet un arbre bien fondé peut posséder un nœud ayant une infinité de fils, et la borne supérieure peut être alors infinie dans l'équation précédente. À l'aide de cette notion de hauteur, on montre que l'ensemble $\Delta_\alpha(\varphi)$ est constitué des jugements validés par au moins une preuve de hauteur inférieure à α : on dit que ces jugements admettent une complexité inférieure à α. Cette caractérisation des itérés à partir des preuves est classique et se trouve par exemple dans l'article d'Aczel [6, pp. 743, 747].
Pour le principe co-inductif, l'induction transfinie sur les itérés s'interprète plus facilement si l'on raisonne par dualité. Soit $\widetilde{\varphi}$ l'opérateur dual de φ, défini par

$$\widetilde{\varphi}(Z) = \neg\varphi(\neg Z)\,,$$

où pour toute partie Z de l'ensemble des jugements, $\neg Z$ est son complémentaire. Cet opérateur permet de définir le *système dual* de Φ, noté $\widetilde{\Phi}$. Étant donné un itéré $\nabla_\alpha(\varphi)$, on cherche à caractériser ses jugements par les preuves qu'ils admettent dans un certain système d'inférence. Comme l'itéré contient l'ensemble engendré co-inductivement par Φ, ce système est nécessairement une extension de Φ : le plus simple est d'y ajouter des axiomes. La question est de savoir quels axiomes. On peut montrer facilement que le plus grand point fixe de φ a pour complémentaire le plus petit point fixe de $\widetilde{\varphi}$, de même que $\nabla_\alpha(\varphi)$ a pour complémentaire $\Delta_\alpha(\widetilde{\varphi})$, soit l'ensemble des jugements de complexité inférieure à α dans le système dual $\widetilde{\Phi}$. On montrera qu'il suffit

d'ajouter comme axiomes à Φ les jugements de complexité $\alpha+1$ dans $\widetilde{\Phi}$ pour obtenir la caractérisation souhaitée, qui semble nouvelle : l'itéré $\nabla_\alpha(\varphi)$ est alors égal à l'ensemble engendré co-inductivement par cette extension de Φ. Quant à la condition de densité, toujours pour le principe co-inductif, elle permet de définir pour chaque jugement j de Z une équation récursive définissant la preuve de j en fonction de preuves d'autres jugements de Z et de preuves de jugements dans $\mathrm{gfp}(\varphi)$. Plus généralement, étant donné un système d'équations récursives définissant des preuves, pour déduire que Z est inclus dans le plus grand point fixe de φ, il suffit de pouvoir associer à chaque jugement de Z une équation de manière à ce que la valeur de la solution en l'inconnue correspondante donne la preuve du jugement . La condition de densité s'exprime donc plutôt ainsi :

$$\exists Y\,.\; Z \subseteq Y \wedge Y \subseteq \varphi(Y)\,.$$

Par exemple, Y peut être choisi égal à l'ensemble des jugements validés par les preuves solutions du système, ainsi que par leurs sous-preuves. Cette condition est seulement en apparence plus faible que la condition donnée précédemment, car toutes les deux sont équivalentes à la conclusion du principe co-inductif, $Z \subseteq \mathrm{gfp}(\varphi)$.
La condition de densité est la condition duale de celle de stabilité. Elle est particulièrement utilisée pour montrer que deux processus sont bisimilaires, habituellement sous sa forme plus forte, $Z \subseteq \varphi(Z)$. Illustrons notre comparaison par cet exemple de la bisimilarité, au cœur de la sémantique des processus communicants depuis les travaux fondateurs de Park [63] et Milner [54].

L'exemple de la bisimilarité entre processus Considérons un ensemble de processus, généralement donné par une algèbre, et une relation de transitions étiquetées entre processus. Si P et P' sont deux processus, la transition $P \xrightarrow{a} P'$ signifie que le processus P peut se transformer en P' en réalisant l'action (observable) a. Deux *processus* sont dits *bisimilaires* s'ils réalisent exactement les mêmes actions, plus exactement les mêmes séquences d'actions maximales : ce n'est pas leur structure interne qui importe, mais leurs actions observables, d'où le terme de simulation pour rendre compte de cette équivalence. Formellement, la relation de bisimilarité est définie par un système d'inférence. Pour faciliter l'expression des règles d'inférence, il est utile d'appeler une fonction de choix pour un processus P toute fonction S de l'ensemble des actions dans l'ensemble des processus telle que pour toute action a du domaine de S, la transition $P \xrightarrow{a} S(a)$ est

possible. Le système d'inférence contient alors toutes les règles bien définies de la forme suivante, pour tout couple de processus (P_1, P_2) et toute paire de fonctions de choix S_1 et S_2 pour les processus P_1 et P_2 respectivement :

$$\frac{\{(P_1', S_2(a)) \mid P_1 \xrightarrow{a} P_1'\} \cup \{(S_1(a), P_2') \mid P_2 \xrightarrow{a} P_2'\}}{(P_1, P_2)}.$$

Comme les processus ont pour vocation d'être des objets au comportement infini, c'est l'interprétation co-inductive de ce système qui est retenue. L'opérateur d'inférence associé φ est alors défini pour toute relation R par :

$$\begin{aligned}\varphi(R) \stackrel{def}{=} \{(P_1, P_2) \mid & \forall\, a\,.\ \forall\, P_1'\,.\ P_1 \xrightarrow{a} P_1' \Rightarrow \\ & \qquad \exists\, P_2'\,.\ P_2 \xrightarrow{a} P_2' \wedge (P_1', P_2') \in R \\ & \wedge \\ & \forall\, P_2'\,.\ P_2 \xrightarrow{a} P_2' \Rightarrow \\ & \qquad \exists\, P_1'\,.\ P_1 \xrightarrow{a} P_1' \wedge (P_1', P_2') \in R\}\,.\end{aligned}$$

La définition habituelle de la bisimilarité utilise cet opérateur, puisqu'elle est définie comme la plus grande relation R vérifiant $R \subseteq \varphi(R)$ (voir par exemple l'exposé de Milner présentant la sémantique des processus concurrents [55, pp. 1216-1217]). On reconnaît la caractérisation de Tarski du plus grand point fixe de φ. La méthode de démonstration par bisimulation, très utilisée pour les algèbres de processus, déduit de la condition forte de densité la bisimilarité : étant donnée une relation R entre processus, si R est inclus dans $\varphi(R)$, alors les couples de processus dans R sont bisimilaires. À ce principe qui n'est pas optimal, une amélioration peut être apportée, permettant d'utiliser une condition plus faible que la densité. L'article de Sangiorgi [75] en donne une formulation synthétique : étant donnée une relation R, si R est inclus dans $\varphi(F(R))$, pour une application F vérifiant certaines conditions, alors les couples de processus dans R sont bisimilaires. Les conditions portant sur F visent à assurer que la réunion des itérés de R par F, $\bigcup_{i\in\mathbb{N}} F^i(R)$[15], forme une partie dense pour φ (cf. [75, Th. 2.11]). On reconnaît cette fois-ci une instance de la condition $\exists\, S\,.\ R \subseteq S \wedge S \subseteq \varphi(S)$, aboutissant à un principe optimal.

Ici s'achève la présentation du chapitre. Par la suite, nous étudions successivement

– les systèmes d'équations récursives,

[15] On traite ici le seul cas où F est expansive ($R \subseteq F(R)$ pour toute relation R), cas auquel il est possible de se ramener.

– la caractérisation déductive de l'induction et de la co-induction, faisant appel aux preuves.

Nous présentons aussi un exemple élémentaire d'application de l'approche déductive, en définissant l'ordre d'approximation et l'égalité entre arbres par des systèmes d'inférence interprétés co-inductivement. On obtient ainsi une formulation élégante de ces relations habituellement présentées sous la forme d'une extension aux arbres des relations sur les étiquettes (cf. par exemple l'article de Courcelle [21, p. 45]). Nous donnons ces définitions en raison, non pas de leur intérêt intrinsèque, mais de leur utilité dans les chapitres suivants.

Préalablement à tout développement, précisons quelques notations concernant les familles.

Notations : applications, fonctions et familles Une *application* est définie par un ensemble de départ, E, un ensemble d'arrivée, F, et un graphe, relation fonctionnelle incluse dans $E \times F$ et de domaine l'ensemble de départ E. On note F^E l'ensemble des applications de E dans F.

De manière classique, on considère par la suite qu'une *fonction* f de E dans F est le prolongement d'une application définie sur une partie de E, qui associe à tout élément hors de cette partie la valeur de non définition $\perp$. En notant $F_\perp$ l'ensemble (pointé) $F \cup \{\perp\}$, on obtient que toute fonction de E dans F devient une application de E dans $F_\perp$; assez naturellement, l'ensemble des fonctions de E dans F est noté $(F_\perp)^E$. On note $\operatorname{dom} f$ le domaine de f, soit l'ensemble des x de E tels que $f(x) \neq \perp$.

On appelle aussi *famille* le graphe d'une application. Ce faisant, on met l'accent sur une notation particulière : si f est une application de E dans F, on note $(f(i))_{i \in E}$, ou $(f_i)_{i \in E}$, ou encore laconiquement (f), la famille associée. L'ensemble E est appelé l'ensemble des indices de la famille, l'ensemble $f(E)$, celui des éléments de la famille ou l'image de la famille. Il convient de noter que l'ensemble des applications de E dans F est en bijection avec l'ensemble des familles ayant pour ensemble d'indices E et pour image une partie de F ; par la suite, nous utilisons tacitement cette bijection.

Souvent, une famille sert à décrire un ensemble, sous une forme indexée : une famille décrit l'ensemble E si son image est l'ensemble E. Lorsque la famille est injective, c'est-à-dire lorsque l'application associée l'est, on obtient une description univoque de l'ensemble E. Par exemple, la famille identique $(i)_{i \in E}$ décrit l'ensemble E de manière univoque.

Dans le cas où les éléments de la famille sont traités comme des ensembles, on préfère une notation ensembliste. La famille $(A_i)_{i \in I}$ est alors notée comme

le produit cartésien indexé $\prod_{i\in I} A_i$; c'est qu'en effet les familles d'ensembles permettent de généraliser le produit cartésien, une famille $(a_i)_{i\in I}$ appartenant au produit cartésien généralisé $\prod_{i\in I} A_i$ si pour tout i, on a $a_i \in A_i$.
Enfin, évoquons quelques familles remarquables.
Lorsque l'ensemble des indices I est fini, il est implicitement supposé que I est un segment initial de l'ensemble des entiers naturels. Si I a n éléments, on utilise alors une notation cartésienne, $(f_0, \ldots, f_{n-1})$, pour la famille $(f_i)_{0\leq i<n}$; avec le point de vue ensembliste, on note $A_0 \times \ldots \times A_{n-1}$ la famille $(A_i)_{0\leq i<n}$; bien noter que la classe des familles d'ensembles indexées par un segment initial des entiers naturels est en bijection avec la classe des produits cartésiens d'ensembles ; par la suite, là encore, nous utilisons tacitement cette bijection.
La famille associée à la fonction de domaine vide est notée ()[16] ; on associe à l'application constante sur E de valeur c la famille constante $(c)_{i\in E}$; avec le point de vue ensembliste, cette famille se note c^E, puisqu'elle correspond à l'ensemble des applications de E dans c ; la famille associant y à x est notée $(x : y)$; étant donné une famille $(f_i)_{i\in I}$, son prolongement en x n'appartenant pas à I par la valeur y est noté $((f), x : y)$.

1.1 Définir des arbres

Par la suite, les arbres sont représentés de manière concrète. On considère deux ensembles, $\mathcal{P}$, l'alphabet à partir duquel est engendré librement l'ensemble des positions, et $\mathcal{F}$, l'ensemble des étiquettes associées aux positions ; un arbre est alors une fonction de l'ensemble $\mathcal{P}^*$ des positions dans l'ensemble des étiquettes, dont le domaine est fermé par préfixe. On se référera au tableau 1.1 pour des définitions classiques concernant les arbres, qui peuvent être infinis en profondeur et en largeur[17].
Nous plongeons l'ensemble de variables $\mathcal{X}$ dans $\mathcal{T}^\infty_\mathcal{F}[\mathcal{X}]$: à chaque variable x est associé l'arbre $(\varepsilon \mapsto x)$, qu'on notera aussi x. De même, l'ensemble $\mathcal{F}$ est plongé dans $\mathcal{T}^\infty_\mathcal{F}$, tout comme la valeur de non-définition, $\perp$, dans $(\mathcal{T}^\infty_\mathcal{F})_\perp$.
Les notations pour les images de ces plongements sont cohérentes avec la notation préfixe de la fin du tableau 1.1.

[16] Nous utiliserons cette notation également dans l'interprétation ensembliste, à la place de la notation courante 1, ensemble singleton, qui est un élément neutre du produit cartésien (à un isomorphisme près).
[17] C'est-à-dire de degré infini.

$\mathcal{P}^*$ (ensemble des positions) : monoïde libre engendré par $\mathcal{P}$, un mot p est noté $p(0)\dots p(|p|-1)$, ε est le mot vide
$\prec_{\mathrm{p}}$ (relation préfixe immédiate) : $p \prec_{\mathrm{p}} q$ s'il existe $j \in \mathcal{P}$ tel que $p\,j = q$
$\prec_{\mathrm{p}}^+$, $\prec_{\mathrm{p}}^*$ (relations préfixes) : fermetures respectivement transitive, réflexive et transitive de $\prec_{\mathrm{p}}$
$\prec_{\mathrm{s}}$ (relation suffixe immédiate) : $p \prec_{\mathrm{s}} q$ s'il existe $j \in \mathcal{P}$ tel que $j\,p = q$
$\prec_{\mathrm{s}}^+$, $\prec_{\mathrm{s}}^*$ (relations suffixes) : fermetures respectivement transitive, réflexive et transitive de $\prec_{\mathrm{s}}$
$\mathcal{F}$: ensemble des étiquettes
$\bot$: valeur de non-définition, et particulièrement notation pour l'arbre vide
$\mathcal{T}_{\mathcal{F}}^{\infty}$ (arbres étiquetés par $\mathcal{F}$) : une fonction $a : \mathcal{P}^* \to \mathcal{F}$ appartient à $\mathcal{T}_{\mathcal{F}}^{\infty}$ si son domaine, $\operatorname{dom} a$, est non vide et fermé par préfixe
$(\mathcal{T}_{\mathcal{F}}^{\infty})_{\bot} : \mathcal{T}_{\mathcal{F}}^{\infty} \cup \{\bot\}$ (ajout de l'arbre vide)
$\mathcal{X}$: ensemble de variables
$\mathcal{T}_{\mathcal{F}}^{\infty}[\mathcal{X}]$: arbres étiquetés par $(\mathcal{F} \cup \mathcal{X})$ tels que toute variable apparaît nécessairement en une feuille
sous-arbre : $a_{/p}$ est le sous arbre en position p (vide si $p \notin \operatorname{dom} a$)
$\prec$ (relation « est un sous-arbre immédiat de ») : $a \prec b$ s'il existe $j \in \mathcal{P}$ tel que $a = b_{/j}$
$\prec^+$, $\prec^*$ (relations « est un sous-arbre de ») : fermetures respectivement transitive, réflexive et transitive de $\prec$
$f(a_i)_{i\in I}$: notation préfixe pour un arbre $a \in \mathcal{T}_{\mathcal{F}}^{\infty}$ (ou $(\mathcal{T}_{\mathcal{F}}^{\infty})_{\bot}$) défini par $a(\varepsilon) = f$ et pour tout $i \in I$, $a_{/i} = a_i$, avec $a_i \in \mathcal{T}_{\mathcal{F}}^{\infty}$ (ou $(\mathcal{T}_{\mathcal{F}}^{\infty})_{\bot}$) ($I \subseteq \mathcal{P}$)

Tab. 1.1 – Notations pour les arbres

1.1.1 Récurrence et récursion

En premier lieu, nous présentons la méthode qui va nous permettre de raisonner sur les arbres infinis en profondeur, c'est-à-dire possédant un chemin infini à partir de la racine. Rappelons préalablement la manière dont on raisonne sur les arbres bien fondés. Un arbre $a \in \mathcal{T}_{\mathcal{F}}^{\infty}$ est *bien fondé* s'il n'existe pas de suite strictement croissante dans son domaine ordonné par la relation préfixe $\prec_{\mathrm{p}}^{+}$. La relation « est un sous-arbre de », $\prec^{+}$, restreinte à l'ensemble $\mathcal{T}$ des arbres bien fondés, est alors bien fondée : en effet, remarquons que $a_{/p\,q} = (a_{/p})_{/q}$, si bien qu'une suite strictement décroissante d'arbres correspondrait à une suite strictement croissante de positions appartenant au domaine du premier arbre de la suite. La récurrence structurelle est alors l'outil naturel pour montrer des propriétés concernant les arbres bien fondés, et la récursion pour définir des applications de domaine l'ensemble de ces arbres.
Pour les arbres infinis, ces techniques ne sont plus possibles. Cependant, par une forme de dualité, plutôt que de s'appuyer sur la bonne fondation des arbres, on peut raisonner par récurrence sur les positions : on change de perspective, qui d'orientée initialement vers les feuilles se dirige maintenant vers la racine.
Prenons l'exemple d'une propriété définie par un prédicat binaire P, fonction d'un arbre et d'une position. Comme on le verra, d'utiles propriétés sont stables pour un système d'inférence formé de règles de la forme suivante :

$$\frac{\{(a_{/q_i}, r_i) \mid i \in I\}}{(a,p)},$$

où à tout arbre a et à toute position p, a été associée une famille $((q_i, r_i))_{i\in I}$, éventuellement vide, telle que pour tout i appartenant à I, on a $q_i\, r_i = p$ et $q_i \neq \varepsilon$. La stabilité signifie que si pour tout i on a montré $P(a_{/q_i}, r_i)$, alors on peut déduire $P(a,p)$. Elle peut être utilisée de deux manières, duales.
Si on considère les arbres bien fondés, on peut raisonner par récurrence structurelle ainsi. Soit a un arbre et supposons

$$\forall\, b \prec^{+} a\,.\ \forall\, p\,.\ P(b,p)\,.$$

Par stabilité, on déduit que pour toute position p, on a $P(a,p)$, et on peut conclure que la propriété est vérifiée universellement.
Si on considère tous les arbres, bien fondés ou non, on raisonne par récurrence sur les positions, ordonnés par la relation suffixe $\prec_{\mathrm{s}}^{+}$. Soit p une position et supposons

$$\forall\, r \prec_{\mathrm{s}}^{+} p\,.\ \forall\, a\,.\ P(a,r)\,.$$

Par stabilité, on déduit que pour tout arbre a, on a $P(a,p)$, et on peut conclure que la propriété est vérifiée universellement.
Il s'avère que les deux techniques que nous venons d'évoquer sont des instances du théorème classique suivant, valable pour tout ensemble muni d'une relation bien fondée.

1.1.1 Théorème (Récurrence et récursion)
Soit W un ensemble muni d'une relation bien fondée $\sqsubset$, c'est-à-dire n'admettant pas de suite strictement décroissante suivant $\sqsubset$ [18] *:*

$$\forall\, (a_n)_{n\in\mathbb{N}} \in W^{\mathbb{N}}\,.\ \neg(\forall\, n\,.\ a_{n+1} \sqsubset a_n)\,.$$

Considérons une propriété portant sur les éléments de W et définie par un prédicat unaire Q vérifiant :

$$\forall\, a \in W\,.\ (\forall\, a' \sqsubset a\,.\ Q(a')) \Rightarrow Q(a)\,.$$

Alors la propriété est universellement vraie :

$$\forall\, a \in W\,.\ Q(a)\,.$$

Étant donné un ensemble A quelconque, soit F une fonction de $W \times (A_\perp)^W$ dans A telle que pour tout couple (x,f) du domaine de F, f a pour domaine de définition la section initiale ouverte engendrée par x :

$$\forall\, y \in W\,.\ y \sqsubset x \Leftrightarrow y \in \operatorname{dom} f\,.$$

Alors il existe une unique application g vérifiant :

$$\forall\, x \in W\,.\ g(x) = F(x, g_{|\sqsubset x})\,,$$

où $g_{|\sqsubset x}$ est la restriction de g à la section initiale ouverte engendrée par x.

La démonstration que nous donnons utilise un système d'inférence pour construire le graphe de l'application définie récursivement. Il conviendrait donc de lire préalablement les définitions du paragraphe 1.2.1 : outre les définitions d'un système d'inférence, de ses règles d'inférence et des preuves

[18] Il s'agit plutôt d'un critère de terminaison pour la relation inverse de $\sqsubset$. Il est souvent préféré la définition nœthérienne d'une relation bien fondée, qui affirme que toute partie non vide admet un élément minimal suivant $\sqsubset$. Cette définition implique le critère de terminaison, qui lui est équivalent si l'on admet l'axiome du choix. Remarquons que la démonstration du théorème utilise d'ailleurs cet axiome.

dans ce système, il est indispensable de savoir que l'ensemble engendré inductivement par ce système se définit comme l'ensemble des conclusions des preuves qui sont bien fondées.

Démonstration

• Récurrence

Supposons $\forall\, a \in W\,.\ (\forall\, a' \sqsubset a\,.\ Q(a')) \Rightarrow Q(a)$. Montrons par l'absurde le résultat. On suppose que le complémentaire de Q dans W n'est pas vide ; on va construire par récurrence une suite $(a_n)_n$ formée d'éléments du complémentaire de Q, strictement décroissante suivant $\sqsubset$, ce qui contredit la bonne fondation de W.

◦ $n = 0$

Par hypothèse, il existe a n'appartenant pas à Q et on pose $a_0 = a$.

◦ $n > 0$

Supposons la suite construite jusqu'au rang $n-1$. Nécessairement, il existe $a' \sqsubset a_{n-1}$ n'appartenant pas à Q et on pose $a_n = a'$.

• Récursion

L'unicité se démontre par récurrence structurelle sur W sans difficulté.

Montrons l'existence. Considérons le système d'inférence défini sur $W \times A$ et composé uniquement des règles

$$\frac{\{(y, f(y)) \mid y \sqsubset x\}}{(x, F(x, f))},$$

pour tout x dans W et toute fonction f de W dans A de domaine $\sqsubset x$.

Ce système a pour but de décrire le graphe de g. Considérons l'ensemble engendré inductivement par ce système, I, et montrons qu'il décrit un graphe d'application.

Tout d'abord, montrons que c'est un graphe de fonction : si (x, a) et (x, a') sont dans I, alors $a = a'$.

C'est immédiat par récurrence structurelle sur la preuve bien fondée de (x, a).

Montrons maintenant que I définit un graphe d'application.

On montre par récurrence sur W la propriété suivante : pour tout x de W, il existe un élément a de A tel que (x, a) appartienne à I.

Soit x dans W. Supposons que pour tout $y \sqsubset x$, il existe a_y tel que (y, a_y) appartienne à I. Alors soit f la fonction définie sur $\sqsubset x$ par $f(y) = a_y$. Définissons a par $F(x, f)$: on a bien (x, a) dans I.

Finalement, par construction, l'application g définie inductivement par le système d'inférence vérifie $\forall\, x \in W\,.\ g(x) = F(x, g_{|\sqsubset x})$.

⌞

Par la suite, on utilisera ce théorème pour l'ensemble des arbres bien fondés et pour l'ensemble des positions, le monoïde libre $\mathcal{P}^*$, bien fondé suivant la relation d'ordre suffixe $\prec_{\mathrm{s}}^{+}$.

1.1.2 Résoudre des systèmes d'équations récursives

Venons-en maintenant au cœur de cette partie, et définissons les systèmes d'équations récursives que nous considérons.

En un premier temps, nous considérons l'ensemble $\mathcal{T}_{\mathcal{F}}^{\infty}$ des arbres étiquetés ; dans un second temps, on construit les arbres en respectant une contrainte supplémentaire, exprimée sous la forme d'une signature.

Pour matérialiser la solution d'un système, introduisons les valuations. Une valuation attribue à chaque variable une valeur, choisie parmi les arbres sans variable, et lorsqu'elle est appliquée à un arbre possédant des variables, elle remplace chacune de ses variables par la valeur associée. Précisément, une *valuation* ν est une application de $\mathcal{X}$ dans $\mathcal{T}_{\mathcal{F}}^{\infty}$. Son extension, de $\mathcal{T}_{\mathcal{F}}^{\infty}[\mathcal{X}]$ dans $\mathcal{T}_{\mathcal{F}}^{\infty}$, notée $[\nu]$ en position suffixe, est définie ainsi, pour tout arbre $a \in \mathcal{T}_{\mathcal{F}}^{\infty}[\mathcal{X}]$ et toute position p :

$$a[\nu](p) \stackrel{def}{=} \begin{cases} a(p) & \text{si } a(p) \in \mathcal{F}\,, \\ \nu(y)(r) & \text{si } \begin{cases} p = q\,r\,, \\ a(q) = y \quad (y \in \mathcal{X})\,, \end{cases} \\ \bot & \text{sinon.} \end{cases}$$

Remarquons que dans le second cas, la décomposition de p en $q\,r$ est unique, tout comme la variable y, puisqu'une variable ne peut être qu'une feuille.
Un *système d'équations récursives* sur $\mathcal{T}_{\mathcal{F}}^{\infty}[\mathcal{X}]$ est une famille (non nécessairement finie) $(a_x)_{x\in\mathcal{X}}$ d'arbres appartenant à $\mathcal{T}_{\mathcal{F}}^{\infty}[\mathcal{X}]$, indexée par l'ensemble $\mathcal{X}$ des variables inconnues, et est noté ainsi :

$$(x = a_x)_{x\in\mathcal{X}}\,.$$

Pour toute variable x, l'expression $x = a_x$ est alors appelée l'*équation* en x du système ; elle est *gardée* si son membre droit possède une étiquette de garde :

$$a_x(\varepsilon) \in \mathcal{F}\,.$$

Le système est *gardé* si chacune de ses équations est gardée. Une *solution* du système est une valuation ν telle que :

$$\forall\, x \in \mathcal{X}\,.\ \nu(x) = a_x[\nu]\,.$$

Souvent, nous considérerons des systèmes *quasi-uniformes*, qui sont gardés : soit a_x appartient à $\mathcal{T}^\infty_\mathcal{F}$, soit $a_x = f(x_i)_{i\in I}$ (en notation préfixe), où f appartient à $\mathcal{F}$ et pour tout i, x_i est une variable.
Le théorème suivant est fondamental.

1.1.2 Théorème (Existence et unicité de la solution dans $\mathcal{T}^\infty_\mathcal{F}$)
Tout système gardé d'équations récursives sur $\mathcal{T}^\infty_\mathcal{F}[\mathcal{X}]$ admet une unique solution à valeur dans $\mathcal{T}^\infty_\mathcal{F}$.

Démonstration
Soient $(x = a_x)_{x\in\mathcal{X}}$ un système gardé et ν une valuation. On a :

$$\forall\, x \in \mathcal{X}\,.\ \nu(x) = a_x[\nu] \quad\Leftrightarrow\quad \forall\, x \in \mathcal{X}, p \in \mathcal{P}^*\,.\ \nu(x)(p) = a_x[\nu](p)\,.$$

Évaluons $a_x[\nu](p)$:

$$a_x[\nu](p) = \begin{cases} a_x(p) & \text{si } a_x(p) \in \mathcal{F}\,, \\ \nu(y)(r) & \text{si } \begin{cases} p = q\,r\,, \\ a_x(q) = y \quad (y \in \mathcal{X})\,, \end{cases} \\ \bot & \text{sinon.} \end{cases}$$

Puisque le système est gardé, dans le second cas, $r \prec^+_\mathrm{s} p$. Ainsi, la valuation ν est solution si et seulement si la famille $(\nu(x))_x$ vérifie une définition récursive sur l'ensemble bien fondé des positions (ordonné suivant la relation suffixe). Par le théorème 1.1.1 (p. 47), il existe une unique famille vérifiant cette définition récursive : c'est l'unique solution du système.
⌞

Plutôt que seulement considérer des arbres étiquetés, il est souvent utile de munir l'ensemble des arbres d'une structure algébrique. Chaque étiquette symbolise alors une fonction possédant un certain profil. Si les arbres sont d'une seule sorte, le profil se réduit à l'arité, soit le nombre d'arguments ; s'ils sont de plusieurs sortes, le profil se compose de l'arité, qui donne les sortes du premier argument jusqu'au dernier, ainsi que de la sorte du résultat. Dans les deux cas, le nombre d'arguments est fini, et cette limitation nous empêche de considérer les arbres étiquetés comme un cas particulier d'arbres respectant une certaine signature. Aussi, nous généralisons cette notion de signature, de manière à pouvoir traiter par la suite le seul cas des arbres respectant une signature sans restreindre la portée de notre propos : nous parlerons de *signature concrète*, dans le sens où l'ensemble des positions $\mathcal{P}^*$ est pris en compte dans la définition de la signature.

On considère un ensemble de sortes $\mathcal{S}$, un ensemble $\mathcal{P}$ engendrant librement les positions et un ensemble d'étiquettes $\mathcal{F}$.
Une signature concrète définie sur $\mathcal{S}$, $\mathcal{P}$ et $\mathcal{F}$ est formée de deux applications, d'une part d'une application de $\mathcal{F}$ dans l'ensemble des fonctions de $\mathcal{P}$ dans $\mathcal{S}$ donnant pour chaque étiquette son arité concrète, soit la position et la sorte de chaque argument, et d'autre part d'une application de $\mathcal{F}$ dans $\mathcal{S}$, donnant la sorte du résultat.
Soit $\Sigma = (\mathbf{A} : \mathcal{F} \to (\mathcal{S}_\perp)^{\mathcal{P}}, \mathbf{R} : \mathcal{F} \to \mathcal{S})$ une signature concrète. Si f est une étiquette, le couple $(\mathbf{A}(f), \mathbf{R}(f))$ est appelé le profil de f, et se note

$$\Big(\prod_{j \in \operatorname{dom} \mathbf{A}(f)} \mathbf{A}(f)(j) \Big) \to \mathbf{R}(f)\,.$$

Un arbre a appartenant à $\mathcal{T}_{\mathcal{F}}^{\infty}$ respecte la signature Σ si pour toute position p de son domaine, la condition d'arité est vérifiée :

$$\forall\, j \in \mathcal{P}, s \in \mathcal{S}\,.\ \mathbf{A}(a(p))(j) = s \Leftrightarrow (p\,j \in \operatorname{dom} a \wedge \mathbf{R}(a(p\,j)) = s)\,.$$

Ainsi, dans le cas classique d'une signature à une seule sorte, qu'on notera s, où l'arité d'une étiquette se définit par le nombre de ses arguments, il est possible de prendre pour $\mathcal{P}$ l'ensemble des entiers naturels, $\mathbb{N}$, et de définir la signature concrète ainsi : si une étiquette a pour arité n, alors son arité concrète est la fonction qui associe à tout entier entre 0 et $n-1$ la sorte s, la sorte de son résultat étant évidemment s ; dans ce cas, la condition d'arité se récrit ainsi, pour un arbre a en une position p de son domaine telle que $a(p)$ a pour arité n :

$$\forall\, j \in \mathcal{P}\,.\ 0 \leq j < n \Leftrightarrow p\,j \in \operatorname{dom} a\,.$$

Comme annoncé en introduction, montrons que l'ensemble $\mathcal{T}_{\mathcal{F}}^{\infty}$ des arbres étiquetés est isomorphe à l'ensemble $\mathcal{T}_{\Sigma}^{\infty}$ des arbres respectant une certaine signature concrète Σ, à une seule sorte s. Cet isomorphisme qu'on note ι annote chaque étiquette d'un arbre par l'arité concrète rencontrée en la position considérée, sa réciproque les effaçant. Précisément, la signature concrète Σ est définie sur le singleton $\{s\}$, formé de l'unique sorte, l'ensemble $\mathcal{P}$ engendrant librement les positions et sur l'ensemble $\mathcal{F} \times (\{s\}_\perp)^{\mathcal{P}}$ des étiquettes, et est munie des deux applications $\mathbf{A}$ et $\mathbf{R}$ vérifiant

$$\begin{aligned} \mathbf{A}((f, A)) &\stackrel{def}{=} A\,, \\ \mathbf{R}((f, A)) &\stackrel{def}{=} s\,, \end{aligned}$$

alors que l'isomorphisme ι vérifie l'équation :

$$\iota(f(a_i)_{i\in I}) = (f, (s)_{i\in I})(\iota(a_i))_{i\in I}\,.$$

Nous récapitulons dans le tableau 1.2 les définitions et les notations utiles concernant les signatures concrètes et les termes, c'est-à-dire les arbres respectant une signature. Bien noter la différence entre les valeurs de non-définition $\bot_s$ et la valeur de non-définition, $\bot$, vue précédemment. Une position p dans un arbre a telle que $a(p) = \bot_s$ appartient au domaine de a, alors que la valeur de non-définition $\bot$ se définit par l'équivalence suivante :

$$a(p) = \bot \overset{def}{\Leftrightarrow} p \notin \operatorname{dom} a\,.$$

Aussi, l'arbre vide, précédemment défini comme l'arbre de domaine vide et noté $\bot$, ne respecte aucune signature concrète. À la place, pour chaque sorte s, il existe un arbre vide, noté $\bot_s$, de domaine ε. Bien sûr, l'isomorphisme décrit précédemment entre $\mathcal{T}^\infty_{\mathcal{F}}$ et $\mathcal{T}^\infty_{\Sigma}$, pour une certaine signature concrète Σ, se prolonge en un isomorphisme entre $(\mathcal{T}^\infty_{\mathcal{F}})_\bot$ et $\mathcal{T}^\infty_{\Sigma_\bot}$.
Enfin, l'égalité entre deux termes a et b est équivalente à :

$$\forall\, p \in \mathcal{P}^*\,.\ a(p) = b(p)\,,$$

ou encore, sans utiliser la valeur $\bot$,

$$(\operatorname{dom} a = \operatorname{dom} b) \wedge (\forall\, p \in \operatorname{dom} a\,.\ a(p) = b(p))\,.$$

On peut affaiblir cette dernière proposition, grâce à la condition d'arité.

1.1.3 Lemme (Égalité entre termes)
Considérons une signature concrète Σ et deux Σ-termes a et b. Alors :

$$a = b \Leftrightarrow (\operatorname{dom} a \subseteq \operatorname{dom} b) \wedge (\forall\, p \in \operatorname{dom} a\,.\ a(p) = b(p))\,.$$

Démonstration
Supposons $\operatorname{dom} a \subseteq \operatorname{dom} b$ et $\forall\, p \in \operatorname{dom} a\,.\ a(p) = b(p)$. Montrons que $\operatorname{dom} a = \operatorname{dom} b$ par l'absurde.
Soit p tel que $a(p) = \bot$ et $b(p) \neq \bot$.
Considérons l'ensemble P formé des préfixes q de p vérifiant $a(q) \neq \bot$:

$$P = \{q \prec^*_{\mathrm{p}} p \mid a(q) \neq \bot\}\,.$$

Comme a n'est pas l'arbre vide, P n'est pas vide. Soit q le plus grand élément de P pour la relation d'ordre $\prec^*_{\mathrm{p}}$; nécessairement $q \prec^+_{\mathrm{p}} p$. Comme $b(p) \neq \bot$,

$\mathcal{S}$: ensemble des sortes
$\mathcal{P}^*$: ensemble des positions
$\mathcal{F}$: ensemble des étiquettes
Σ (signature concrète) :
$\mathbf{A} : \mathcal{F} \to (\mathcal{S}_\bot)^{\mathcal{P}}$ étiquette $\mapsto$ positions et sortes de ses arguments
$\mathbf{R} : \mathcal{F} \to \mathcal{S}$ étiquette $\mapsto$ sorte du résultat

$$f : (\prod_{j \in \mathrm{dom}\, A} A(j)) \to s \overset{def}{\Leftrightarrow} \mathbf{A}(f) = A \wedge \mathbf{R}(f) = s$$

$\mathcal{T}_\Sigma^\infty$: Σ-algèbre des Σ-termes (arbres bien fondés ou non respectant Σ)

$$\mathcal{T}_\Sigma^\infty \overset{def}{=} \{a \in \mathcal{T}_\mathcal{F}^\infty \quad | \quad \forall\, p \in \mathrm{dom}\, a\,.\ \forall\, j \in \mathcal{P}, s \in \mathcal{S}\,. \\ \mathbf{A}(a(p))(j) = s \Leftrightarrow p\, j \in \mathrm{dom}\, a \wedge \mathbf{R}(a(p\, j)) = s\}$$

$(\mathcal{T}_\Sigma^\infty : s)$: ensemble des Σ-termes de sorte s

$$(\mathcal{T}_\Sigma^\infty : s) \overset{def}{=} \{a \in \mathcal{T}_\Sigma^\infty \mid \mathbf{R}(a(\varepsilon)) = s\}$$
$$a : s \overset{def}{\Leftrightarrow} a \in (\mathcal{T}_\Sigma^\infty : s)$$

$\bot_s$: valeur de non-définition, de profil $() \to s$ (pas d'argument, résultat de sorte s)
$\Sigma_\bot$ (signature pointée) : signature concrète obtenue en ajoutant les valeurs de non-définition $\bot_s$ (pour chaque sorte s)
$\mathcal{T}_{\Sigma_\bot}^\infty$: $\Sigma_\bot$-algèbre des Σ-termes partiels (arbres respectant $\Sigma_\bot$)
$\mathcal{X}_s$: ensemble de variables, de profil $() \to s$ (pas d'arguments, résultat de sorte s)
$\mathcal{X}$: réunion disjointe des ensembles de variables de chaque sorte

$$\mathcal{X} = \sum_{s \in \mathcal{S}} \mathcal{X}_s$$

$\Sigma[\mathcal{X}]$ (signature étendue sur $\mathcal{X}$) : signature concrète obtenue en ajoutant les variables
$\mathcal{T}_\Sigma^\infty[\mathcal{X}]$: $\Sigma[\mathcal{X}]$-algèbre des Σ-termes ouverts (arbres respectant $\Sigma[\mathcal{X}]$)
$f(a_i)_{i \in I}$: notation préfixe pour un arbre $a \in \mathcal{T}_\Sigma^\infty$ (ou $\mathcal{T}_{\Sigma_\bot}^\infty$) défini par $a(\varepsilon) = f$ et pour tout $i \in I$, $a_{/i} = a_i$, avec $a_i \in \mathcal{T}_\Sigma^\infty$ (ou $\mathcal{T}_{\Sigma_\bot}^\infty$) ($I \subseteq \mathcal{P}$)
$f(a_0, \ldots, a_{i-1})$: notation préfixe pour le même arbre $a \in \mathcal{T}_\Sigma^\infty$ (ou $\mathcal{T}_{\Sigma_\bot}^\infty$) (cas classique où $\mathcal{P} = \mathbb{N}$ et où les arguments sont en nombre fini et classés de 0 à $i - 1$)

Tab. 1.2 – Notations pour les termes

alors $b(q) \neq \bot$ et il existe j appartenant à $\mathcal{P}$ et une sorte s tels que $q\,j \prec_{\mathrm{p}}^{*} p$, $q\,j \in \operatorname{dom} b$ et $\mathbf{A}(b(q))(j) = s$. Comme $a(q) = b(q)$, on en déduit que $q\,j \in \operatorname{dom} a$ puisque a respecte la signature Σ, ce qui contredit la définition de q. Finalement, $\operatorname{dom} a = \operatorname{dom} b$ et a est bien égal à b.
⌞

Désormais, nous considérons l'ensemble $\mathcal{T}_{\Sigma}^{\infty}$ des Σ-termes pour une signature concrète Σ quelconque. Reprenons la démarche précédente pour la résolution d'équations récursives, de manière à prendre en compte la condition d'arité.
Tout d'abord, les équations doivent respecter les sortes. Ainsi, un *système d'équations récursives* sur $\mathcal{T}_{\Sigma}^{\infty}[\mathcal{X}]$ est une famille (non nécessairement finie) $(a_x)_{x\in\mathcal{X}}$ de Σ-termes ouverts appartenant à $\mathcal{T}_{\Sigma}^{\infty}[\mathcal{X}]$, indexée par l'ensemble $\mathcal{X}$ des variables inconnues, et telle que pour toute inconnue x, la sorte de x est égale à celle de a_x :

$$a_x \,:\, \mathbf{R}(x)\,.$$

Ensuite, une valuation doit aussi respecter les sortes. Une application $\nu : \mathcal{X} \to \mathcal{T}_{\Sigma}^{\infty}$ est une *valuation*, ce qu'on note $\nu : \mathcal{X} \xrightarrow{\mathrm{v}} \mathcal{T}_{\Sigma}^{\infty}$, si pour toute variable x, la sorte de x est égale à celle de $\nu(x)$:

$$\nu(x) \,:\, \mathbf{R}(x)\,.$$

L'extension d'une valuation, telle que définie précédemment, est une application de $\mathcal{T}_{\Sigma}^{\infty}[\mathcal{X}]$ dans $\mathcal{T}_{\mathcal{F}}^{\infty}$. Nous aimerions vérifier que son image est incluse dans $\mathcal{T}_{\Sigma}^{\infty}$. De même, pour la résolution d'équations, nous aimerions montrer que tout système gardé sur $\mathcal{T}_{\Sigma}^{\infty}[\mathcal{X}]$ admet une unique solution à valeur dans $\mathcal{T}_{\Sigma}^{\infty}$. Remarquons que cette dernière propriété suffit à montrer la précédente, concernant les valuations. En effet, soient $\nu : \mathcal{X} \xrightarrow{\mathrm{v}} \mathcal{T}_{\Sigma}^{\infty}$ une valuation et a un Σ-terme ouvert de $\mathcal{T}_{\Sigma}^{\infty}[\mathcal{X}]$; si a est une variable, alors $a[\nu]$ appartient évidemment à $\mathcal{T}_{\Sigma}^{\infty}$, sinon $a[\nu]$ est la composante en y de l'unique solution du système gardé $(y = a, (x = \nu(x))_{x\in\mathcal{X}})$, où y est une nouvelle variable.
Pour la valeur en une inconnue de la solution d'un système, il nous importe donc de vérifier la condition d'arité en toute position. Cette condition constitue un exemple de propriétés particulières, celles qui expriment en toute position une propriété du sous-arbre y commençant. Pour montrer de telles propriétés, dites locales, on procède par une forme simplifiée de récurrence, suivant la construction de la solution.

1.1.4 Définition et proposition (Lemme des propriétés locales)
Une propriété définie par un prédicat binaire I de paramètres un arbre et une position est locale *si elle vérifie pour tout arbre a de $\mathcal{T}_{\mathcal{F}}^{\infty}$ et toute position p*

du domaine de a l'équivalence suivante :

$$I(a,p) \Leftrightarrow I(a_{/p},\varepsilon)\,.$$

Considérons une propriété locale définie par un prédicat I.
Soient $(x = a_x)_{x\in\mathcal{X}}$ un système gardé sur $\mathcal{T}^{\infty}_{\mathcal{F}}[\mathcal{X}]$, et ν son unique solution.
Supposons que pour toute variable x et toute position p appartenant à $\operatorname{dom} a_x$ *telle que $a_x(p) \in \mathcal{F}$, on ait $I(\nu(x),p)$.*
Alors, pour toute variable x de $\mathcal{X}$ et toute position p de $\operatorname{dom}\nu(x)$*, $I(\nu(x),p)$ est vérifié : on dit que la valuation ν a pour* invariant *la propriété locale définie par I.*

Démonstration
Montrons pour toute position p, la propriété

$$\forall\, x\,.\ p \in \operatorname{dom}\nu(x) \Rightarrow I(\nu(x),p)$$

par récurrence sur l'ensemble des positions, muni de la relation bien fondée $\prec_{\mathrm{s}}^{+}$ (la relation suffixe).
Soit p une position et supposons qu'on ait pour toute position r telle que $r \prec_{\mathrm{s}}^{+} p$, la propriété

$$\forall\, x\,.\ r \in \operatorname{dom}\nu(x) \Rightarrow I(\nu(x),r)\,.$$

Distinguons deux cas.
• $p = \varepsilon$
Soit x une variable. On a évidemment $p \in \operatorname{dom}\nu(x)$; montrons donc $I(\nu(x),p)$.
Puisque le système est gardé, $a_x(\varepsilon) \in \mathcal{F}$, d'où $I(\nu(x),\varepsilon)$ par hypothèse.
• $p \neq \varepsilon$
Soit x une variable telle que p appartienne à $\operatorname{dom}\nu(x)$.
Si $a_x(p) \in \mathcal{F}$, alors par hypothèse, on a $I(\nu(x),p)$.
Sinon, $p = q\,r$, avec $a_x(q) = y$, y étant une variable. Par définition de ν, on a $\nu(x)_{/q} = \nu(y)$. On en déduit, grâce au caractère local de I, les équivalences suivantes :

$$\begin{aligned} I(\nu(x),p) &\Leftrightarrow I(\nu(x)_{/p},\varepsilon)\\ &\Leftrightarrow I(\nu(y)_{/r},\varepsilon)\\ &\Leftrightarrow I(\nu(y),r)\,. \end{aligned}$$

Puisque le système est gardé, nous avons $q \neq \varepsilon$, soit $r \prec_{\mathrm{s}}^{+} p$; par l'hypothèse de récurrence, $I(\nu(y),r)$ est vérifié, et donc finalement $I(\nu(x),p)$ aussi.
⌞

Nous sommes maintenant en mesure de montrer le théorème analogue au 1.1.2.

1.1.5 Théorème (Existence et unicité de la solution dans $\mathcal{T}_\Sigma^\infty$)
Tout système gardé d'équations récursives sur $\mathcal{T}_\Sigma^\infty[\mathcal{X}]$ admet une unique solution à valeur dans $\mathcal{T}_\Sigma^\infty$.

Démonstration
Soient $(x = a_x)_{x\in\mathcal{X}}$ un système gardé, où $a_x \in \mathcal{T}_\Sigma^\infty[\mathcal{X}]$, et ν son unique solution à valeur dans $\mathcal{T}_\mathcal{F}^\infty$. Nous définissons une propriété permettant de vérifier l'arité, notée I, à savoir, pour tout arbre a et toute position p de son domaine :

$$I(a,p) \quad \stackrel{def}{\Leftrightarrow} \quad \forall\, j \in \mathcal{P}, s \in \mathcal{S}\,.\ \mathbf{A}(a(p))(j) = s \Leftrightarrow \left(\begin{array}{l} p\,j \in \operatorname{dom} a \\ \mathbf{R}(a(p\,j)) = s \end{array} \right).$$

Montrons que les conditions pour appliquer le lemme des propriétés locales sont remplies par I.
Que le prédicat I définisse une propriété locale est clair.
Soient x une variable et p une position appartenant à $\operatorname{dom} a_x$ telle que $a_x(p) \in \mathcal{F}$. Montrons $I(\nu(x), p)$.
Par définition de la solution ν, et comme $a_x(p) \in \mathcal{F}$, on a :

$$\nu(x)(p\,j) = \begin{cases} a_x(p\,j) & \text{si } a_x(p\,j) \in \mathcal{F}\,, \\ a_y(\varepsilon) & \text{si } a_x(p\,j) = y \quad (y \in \mathcal{X})\,, \\ \bot & \text{sinon.} \end{cases}$$

Pour $j \in \mathcal{P}$ et $s \in \mathcal{S}$, la proposition

$$\nu(x)(p\,j) \neq \bot \wedge \mathbf{R}(\nu(x)(p\,j)) = s$$

est successivement équivalente à :

$$\begin{array}{ll} & (a_x(p\,j) \in \mathcal{F} \wedge \mathbf{R}(a_x(p\,j)) = s) \\ \vee & (\exists\, y \in \mathcal{X}\,.\ a_x(p\,j) = y \wedge \mathbf{R}(a_y(\varepsilon)) = s)\,, \end{array}$$

$$\begin{array}{ll} & (a_x(p\,j) \in \mathcal{F} \wedge \mathbf{R}(a_x(p\,j)) = s) \\ \vee & (\exists\, y \in \mathcal{X}\,.\ a_x(p\,j) = y \wedge \mathbf{R}(y) = s) \quad (\mathbf{R}(y) = \mathbf{R}(a_y(\varepsilon)))\,, \end{array}$$

$$\mathbf{A}(a_x(p))(j) = s \quad (a_x \in \mathcal{T}_\Sigma^\infty[\mathcal{X}])\,,$$

$$\mathbf{A}(\nu(x)(p))(j) = s \quad (\nu(x)(p) = a_x(p))\,.$$

Nous avons ainsi vérifié $I(\nu(x), p)$.
D'après la proposition 1.1.4, ν a pour invariant la propriété d'arité I, et nous pouvons conclure :

$$\forall\, x \in \mathcal{X}\ .\ \nu(x) \in \mathcal{T}_\Sigma^\infty\ .$$

⌞

1.1.3 Des équations avec des opérations

Comme annoncé, nous allons maintenant autoriser des opérations sur les arbres dans les équations, afin de permettre une plus grande souplesse d'utilisation des systèmes d'équations.
On se donne pour la suite une signature concrète Σ, définie sur l'ensemble $\mathcal{S}$ des sortes, l'ensemble $\mathcal{P}$ engendrant librement les positions, et l'ensemble $\mathcal{F}$ des étiquettes, et munie des applications $\mathbf{A} : \mathcal{F} \to (\mathcal{S}_\perp)^\mathcal{P}$, et $\mathbf{R} : \mathcal{F} \to \mathcal{S}$, donnant l'arité et la sorte de chaque étiquette. L'ensemble des variables inconnues utilisées dans les équations est toujours noté $\mathcal{X}$.
Dans tout ce paragraphe, nous illustrerons notre propos par un exemple. On considère dans une signature à une seule sorte une variable x, une étiquette f unaire, l'opération unaire α, ainsi que l'équation

$$x = \alpha(f(\alpha(x)))\,.$$

Il s'agit de résoudre cette équation pour certaines opérations, dites autorisées.
On notera $\mathcal{O}$ l'ensemble des opérations autorisées dans les équations. Un élément de $\mathcal{O}$ sera considéré tantôt comme une application, tantôt comme une étiquette. En tant qu'étiquette, il possède un profil, et en tant qu'application, il possède un type, obtenu en interprétant chaque sorte s par l'ensemble des termes de sorte s, $(\mathcal{T}_\Sigma^\infty : s)$. Précisément, on suppose que l'ensemble $\mathcal{O}$ est à la fois, suivant le point de vue,

- une signature, associant à chaque opération un profil,
- un ensemble d'applications,

tels que si l'opération α a pour profil $\prod_{i\in I} A_i \to s$ ($I \subseteq \mathcal{P}$), alors α est aussi une application de $\prod_{i\in I}(\mathcal{T}_\Sigma^\infty : A_i)$ dans $(\mathcal{T}_\Sigma^\infty : s)$.
Pour lever toute ambiguïté, rappelons une différence entre la notation préfixe pour les arbres et l'application d'une opération à une famille : $\alpha((a_i)_{i\in I})$ représente le résultat de l'application de α à la famille $(a_i)_{i\in I}$, alors que $\alpha(a_i)_{i\in I}$ représente l'arbre de racine α et de sous-arbre a_i en position i, pour tout indice i.

Dans un premier temps, nous allons définir la forme des équations récursives étudiées, puis nous résoudrons des systèmes de telles équations. Cette résolution sera possible si l'ensemble des opérations autorisées vérifie certaines conditions, que nous allons mettre en évidence. On montrera aussi l'existence d'un tel ensemble d'opérations autorisées, particulièrement simple et utile dans la pratique.

Comme nous l'avons vu en introduction, le membre droit d'une équation est un $\Sigma + \mathcal{O}$-terme ouvert d'une forme particulière, $\Sigma + \mathcal{O}$ étant la signature obtenue par réunion de Σ et de $\mathcal{O}$: des contraintes limitent en effet les possibilités d'utilisation des opérations.
Commençons par le cas élémentaire : les opérations autorisées n'apparaissent dans le membre droit qu'en étant appliquées à des variables. Pour représenter les applications des opérations aux variables, on préfère introduire un second ensemble de variables, noté $\mathcal{O}(\mathcal{X}^*)$ et indexé injectivement par l'ensemble des couples $(\alpha, (x_i)_{i\in I})$, où α, appartenant à $\mathcal{O}$, et la famille $(x_i)_{i\in I}$ de variables sont tels que $\alpha(x_i)_{i\in I}$ appartient à $\mathcal{T}^\infty_{\mathcal{O}}[\mathcal{X}]$. On notera $\alpha(x_i)_{i\in I}$ la variable indexée par $(\alpha, (x_i)_{i\in I})$: autrement dit, on remplace dans un membre droit tout sous-arbre $\alpha(x_i)_{i\in I}$ par la variable indexée par $(\alpha, (x_i)_{i\in I})$ et notée aussi $\alpha(x_i)_{i\in I}$.
À partir de ce cas élémentaire, les membres droits se construisent inductivement ; on note $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$ l'ensemble des membres droits admissibles dans les équations avec opérations. Voici les règles d'inférence permettant d'engendrer inductivement l'ensemble $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$:
un arbre a de $\mathcal{T}^\infty_{\Sigma+\mathcal{O}}[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$ appartient à $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$

– si a appartient à $\mathcal{T}^\infty_{\Sigma}[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$:

$$\frac{\emptyset}{a} \quad (a \in \mathcal{T}^\infty_{\Sigma}[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)])\,,$$

– s'il existe une famille $(a_i)_{i\in I}$ d'arbres appartenant à $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$ et une étiquette f telles que a est égal à $f(a_i)_{i\in I}$:

$$\frac{(a_i)_{i\in I}}{f(a_i)_{i\in I}} \quad (f \in \mathcal{F})\,,$$

– s'il existe une famille $(a_i)_{i\in I}$ d'arbres appartenant à $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$ et une opération autorisée α telles que a est égal à $\alpha(a_i)_{i\in I}$:

$$\frac{(a_i)_{i\in I}}{\alpha(a_i)_{i\in I}} \quad (\alpha \in \mathcal{O})\,,$$

ce qu'on résume par la grammaire suivante :

$$\begin{array}{rcll} a & ::= & b & (b \in \mathcal{T}_\Sigma^\infty[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]) \\ & | & f(a_i)_{i\in I} & (f \in \mathcal{F}) \\ & | & \alpha(a_i)_{i\in I} & (\alpha \in \mathcal{O})\,. \end{array}$$

Nous considérons donc des systèmes d'équations récursives de la forme

$$(x = a_x)_{x\in\mathcal{X}}\,,$$

où pour toute inconnue x de $\mathcal{X}$,

- le membre droit a_x appartient à $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$,
- si x a pour sorte s, alors a_x également.

Le système $(x = a_x)_{x\in\mathcal{X}}$ est dit *gardé* si pour toute inconnue x, le membre droit a_x est gardé. Quant à la propriété de garde pour les membres droits, elle se définit inductivement ainsi, pour tout arbre a de $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$:

- $a \in \mathcal{T}_\Sigma^\infty[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$: a est gardé si $a(\varepsilon)$ est une étiquette,
- $a = f(a_i)_{i\in I}$ $(f \in \mathcal{F})$: a est gardé,
- $a = \alpha(a_i)_{i\in I}$ $(\alpha \in \mathcal{O})$: a est gardé si pour tout i de I, a_i est gardé.

Autrement dit, un arbre de $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$ est gardé si toute variable y apparaissant est gardée par une étiquette. Par exemple, si f est une étiquette unaire, α une opération unaire autorisée et x une variable, alors :

- $f(x)$, $\alpha(f(x))$, $f(\alpha(x))$ et $\alpha(f(\alpha(x)))$ sont gardés,
- $\alpha(x)$ n'est pas gardé.

Définissons maintenant la solution d'un tel système d'équations.
Soit $\nu : \mathcal{X} \xrightarrow{\text{v}} \mathcal{T}_\Sigma^\infty$ une valuation. Son extension à $\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)$, toujours notée ν, est définie par :

$$\nu(\alpha(x_i)_{i\in I}) \stackrel{def}{=} \alpha((\nu(x_i))_{i\in I})\,.$$

Il est désormais possible d'étendre ν à l'ensemble des membres droits admissibles, $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$. L'*extension de la valuation*, application de $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$ dans $\mathcal{T}_\Sigma^\infty$, notée $[\nu]$ en position suffixe, se définit inductivement de la manière suivante, pour tout arbre a appartenant à $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$:

- $a \in \mathcal{T}_\Sigma^\infty[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$: $a[\nu]$ est défini de manière classique en toute position p par l'équation suivante :

$$a[\nu](p) \stackrel{def}{=} \begin{cases} a(p) & \text{si } a(p) \in \mathcal{F}\,, \\ \nu(y)(r) & \text{si } \begin{cases} p = q\,r\,, \\ a(q) = y \quad (y \in \mathcal{X})\,, \end{cases} \\ \alpha((\nu(x_i))_{i\in I})(r) & \text{si } \begin{cases} p = q\,r\,, \\ a(q) = \alpha(x_i)_{i\in I} \quad (\alpha(x_i)_{i\in I} \in \mathcal{O}(\mathcal{X}^*))\,, \end{cases} \\ \bot & \text{sinon,} \end{cases}$$

– $a = f(a_i)_{i\in I}$ $(f \in \mathcal{F})$:

$$a[\nu] = f(a_i[\nu])_{i\in I}\,,$$

– $a = \alpha(a_i)_{i\in I}$ $(\alpha \in \mathcal{O})$:

$$a[\nu] = \alpha((a_i[\nu])_{i\in I})\,.$$

Une valuation ν est *solution* du système $(x = a_x)_{x\in\mathcal{X}}$ si pour toute inconnue x, on a :

$$\nu(x) = a_x[\nu]\,.$$

Par exemple, la valuation ν est solution de l'équation

$$x = \alpha(f(\alpha(x)))$$

si elle vérifie[19]

$$\nu(x) = \alpha\big(\big(\, f\big(\, \alpha(\big(\, \nu(x)\,\big)\big)\,\big)\,\big)\big)\,.$$

Nous cherchons à ramener la résolution d'un tel système, admettant des opérations, à celle d'un système classique, pour lequel on dispose d'un théorème d'existence et d'unicité de la solution. À l'aide d'un exemple, dégageons les principes de la résolution.

Pour préciser la définition des opérations autorisées dans les systèmes, un point de vue catégorique est utile pour commencer. On considère

– d'une part, la catégorie ayant pour objets les signatures concrètes et pour morphismes les applications entre ensembles d'étiquettes préservant le profil : si Σ_1 est une signature concrète sur S_1, P_1^* et F_1 définie par $A_1 : F_1 \to ((S_1)_\perp)^{P_1}$ et $R_1 : F_1 \to S_1$, et Σ_2 une signature concrète sur S_2, P_2^* et F_2 définie par $A_2 : F_2 \to ((S_2)_\perp)^{P_2}$ et $R_2 : F_2 \to S_2$, alors l'application $\alpha : F_1 \to F_2$ est un morphisme de Σ_1 vers Σ_2 si $A_1 = A_2 \circ \alpha$ et $R_1 = R_2 \circ \alpha$;
– d'autre part, la catégorie ayant pour objets les ensembles d'arbres respectant une signature concrète et pour morphismes les applications entre ces ensembles.

On peut alors étendre la correspondance entre toute signature concrète Σ et l'ensemble $\mathcal{T}_\Sigma^\infty$ des Σ-termes de manière à obtenir un foncteur entre les catégories associées. Étant donné deux signatures Σ_1 et Σ_2, le foncteur $\mathcal{T}^\infty$ associe à chaque morphisme γ de Σ_1 dans Σ_2, l'application $\mathcal{T}_\gamma^\infty$ de $\mathcal{T}_{\Sigma_1}^\infty$ dans $\mathcal{T}_{\Sigma_2}^\infty$ définie pour tout terme $a \in \mathcal{T}_{\Sigma_1}^\infty$ et toute position p par :

$$\mathcal{T}_\gamma^\infty(a)(p) = \gamma(a(p))\,.$$

[19] Nous préférons conserver l'usage des doubles parenthèses pour l'application, même si une simplification serait bienvenue dans ce cas.

On applique donc à chaque étiquette dans l'arbre l'application γ. Ce genre de foncteur, bien connu, pour les listes par exemple, porte souvent le nom de « map ».
Voyons maintenant la résolution d'un système impliquant une opération obtenue par le foncteur $\mathcal{T}^\infty$. Considérons un morphisme γ de la signature concrète Σ vers elle-même. On obtient par le foncteur $\mathcal{T}^\infty$ une application de $\mathcal{T}^\infty_\Sigma$ dans lui-même, soit $\mathcal{T}^\infty_\gamma$, puis pour chaque sorte s, en restreignant les ensembles de départ et d'arrivée, une application de $\mathcal{T}^\infty_\Sigma : s$ dans lui-même, qu'on note $\mathcal{T}^\infty_\gamma : s$. Cette application, qui a pour profil $s \to s$, sera aussi notée γ_s ; rappelons que son application à un terme a de $\mathcal{T}^\infty_\Sigma : s$ est notée $\gamma_s((a))$, et que le terme de racine l'opération γ_s et de sous-terme immédiat a est noté $\gamma_s(a)$. Supposons que l'ensemble $\mathcal{O}$ contienne l'opération $\mathcal{T}^\infty_\gamma : s$, et qu'il soit fermé par composition à gauche de γ_s (en respectant le typage). Soit $f \in \mathcal{F}$ une étiquette de profil $s \to s$ et considérons le système formé de l'équation

$$x = \gamma_s(f(\gamma_s(x)))\,.$$

On commence par appliquer γ_s, pour obtenir :

$$x = \gamma(f)(\gamma_s^2(x))\,,$$

puis on détermine l'équation vérifiée par $\gamma_s^2(x)$, soit

$$\gamma_s^2(x) = \gamma_s^3(f(\gamma_s(x)))\,,$$

et on peut appliquer γ_s^3, et ainsi de suite.
Pour généraliser cette technique de résolution, il est donc utile d'étendre les opérations de $\mathcal{O}$ aux termes ouverts, c'est-à-dire de les prolonger en opérations sur l'ensemble $\mathcal{T}^\infty_\Sigma[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$.
Voyons comment réaliser cette extension dans le cas de la même opération que précédemment, $\mathcal{T}^\infty_\gamma : s$, notée γ_s. L'application γ, de $\mathcal{F}$ dans $\mathcal{F}$, peut être prolongée en une application de $\mathcal{F} \cup \mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)$ dans lui-même, toujours notée γ, et définie ainsi :

$$\begin{array}{rcll} \gamma(x) & \stackrel{def}{=} & \gamma_s(x) & (x \in \mathcal{X}_s)\,, \\ \gamma(\beta(x_i)_{i\in I}) & \stackrel{def}{=} & (\gamma_s \circ \beta)(x_i)_{i\in I} & \left(\begin{array}{l} \beta(x_i)_{i\in I} \in \mathcal{O}(\mathcal{X}^*) \\ \beta(x_i)_{i\in I} : s \end{array} \right)\,. \end{array}$$

Dans le premier cas, $\gamma_s(x)$ appartient à $\mathcal{O}(\mathcal{X}^*)$ et a pour sorte s, dans le second, grâce à l'hypothèse de fermeture par composition, $(\gamma_s \circ \beta)(x_i)_{i\in I}$ est bien une variable de $\mathcal{O}(\mathcal{X}^*)$, de plus de sorte s. Comme l'application

γ ainsi prolongée préserve les sortes, c'est un morphisme de la signature concrète $\Sigma[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$ vers elle-même ; il s'ensuit que son image par $\mathcal{T}^\infty$, soit $\mathcal{T}^\infty_\gamma$, est une application de $\mathcal{T}^\infty_\Sigma[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$ dans lui-même, définie sans surprise, pour tout terme a et toute position p, par :

$$\mathcal{T}^\infty_\gamma(a)(p) \stackrel{def}{=} \gamma(a(p))\,.$$

Suivant l'exemple, l'idée d'étendre les opérations autorisées en opérations sur $\mathcal{T}^\infty_\Sigma[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$ a été introduite pour transformer l'équation

$$x = \gamma_s(f(\gamma_s(x)))\,,$$

en

$$x = \gamma_s\big(\big(f(\gamma_s(x))\big)\big)\,.$$

L'application de γ_s à $f(\gamma_s(x))$ produit un terme gardé, soit $\gamma(f)(\gamma_s^2(x))$, si bien que la résolution du système transformé est possible. Si on note a le terme $f(\gamma_s(x))$, la solution ν du système transformé est aussi solution du système initial si $\gamma_s((a))[\nu] = \gamma_s((a[\nu]))$, ce qui est le cas si γ_s (son prolongement) commute avec les valuations. On peut donc considérer qu'une telle transformation d'un système est justifiée

- si le système formé des équations transformées, ainsi que des nouvelles équations correspondant aux variables de $\mathcal{O}(\mathcal{X}^*)$, peut être résolu, ce qui est le cas si toute opération autorisée transforme une famille d'arbres gardés en un arbre gardé,
- et si le système transformé et le système original ont la même solution, ce qui est le cas si les opérations autorisées (précisément leur prolongement) commutent avec les valuations.

La préservation de la propriété de garde et la commutativité sont fondamentales pour l'ensemble $\mathcal{O}$, car elles fournissent des conditions suffisantes pour résoudre les systèmes d'équations gardées de manière univoque. Formalisons donc ces propriétés.

1.1.6 Définition (Opérations autorisées)
*Soit $\mathcal{O}$ un ensemble d'opérations sur $\mathcal{T}^\infty_\Sigma$. L'ensemble $\mathcal{O}$ est formé d'*opérations autorisées *dans les systèmes d'équations récursives si chaque opération α de $\mathcal{O}$, définie de $\prod_{i\in I}(\mathcal{T}^\infty_\Sigma : A_i)$ vers $(\mathcal{T}^\infty_\Sigma : s)$, peut être prolongée en une opération de $\prod_{i\in I}(\mathcal{T}^\infty_\Sigma[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)] : A_i)$ vers $(\mathcal{T}^\infty_\Sigma[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)] : s)$, toujours notée α, et vérifiant :*

- *la commutativité avec les valuations :*
  pour toute valuation $\nu : \mathcal{X} \stackrel{\mathrm{v}}{\to} \mathcal{T}^\infty_\Sigma$ et toute famille $(a_i)_{i\in I}$ de termes de

$\mathcal{T}_\Sigma^\infty[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$ *appartenant au domaine de* α,

$$\alpha((a_i)_{i\in I})[\nu] = \alpha((a_i[\nu])_{i\in I})\,,$$ [20]

– *la préservation de la propriété de garde :*
pour toute famille $(a_i)_{i\in I}$ *de termes gardés de* $\mathcal{T}_\Sigma^\infty[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$ *appartenant au domaine de* α, $\alpha((a_i)_{i\in I})$ *est gardé :*

$$\frac{(a_i(\varepsilon) \in \mathcal{F})_{i\in I}}{\alpha((a_i)_{i\in I})(\varepsilon) \in \mathcal{F}}.$$

Il existe au moins un ensemble d'opérations autorisées, l'ensemble des opérations images par le foncteur $\mathcal{T}^\infty$, puisque nous avons le lemme de commutativité suivant.

1.1.7 Lemme (Commutativité entre valuations et opérations fonctorielles)

Soit γ *un morphisme de* Σ *vers* Σ.
Considérons un ensemble $\mathcal{O}$ *d'opérations sur* $\mathcal{T}_\Sigma^\infty$, *tel que pour toute sorte* s, $\mathcal{O}$ *contient l'opération* $\mathcal{T}_\gamma^\infty : s$ *et est fermé par composition à gauche avec* $\mathcal{T}_\gamma^\infty : s$. *Alors, pour toute sorte* s, *pour toute valuation* $\nu : \mathcal{X} \xrightarrow{\text{v}} \mathcal{T}_\Sigma^\infty$ *et tout terme* a *appartenant à* $\mathcal{T}_\Sigma^\infty[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)] : s$, *on a, en notant* γ_s *l'opération* $\mathcal{T}_\gamma^\infty : s$ *:*

$$\gamma_s((a))[\nu] = \gamma_s((a[\nu]))\,.$$

Démonstration
Soient s une sorte, ν une valuation et a un terme de $\mathcal{T}_\Sigma^\infty[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)] : s$. Considérons une position p. On a :

$$\gamma_s((a[\nu]))(p) \;=\; \gamma(a[\nu](p))\,.$$

Examinons les quatre cas relatifs à p intervenant dans le calcul de $a[\nu](p)$.
• $a(p) \in \mathcal{F}$
Dans ce cas, $\gamma_s((a))(p) \in \mathcal{F}$. Vérifions l'égalité :

$$\begin{aligned}\gamma_s((a))[\nu](p) &= \gamma_s((a))(p)\\ &= \gamma(a(p))\,,\\ \gamma_s((a[\nu]))(p) &= \gamma(a(p))\,.\end{aligned}$$

[20] On aurait pu écrire cette égalité ainsi :

$$\alpha((a_i)_{i\in I})[\nu] = \alpha(a_i)_{i\in I}[\nu]\,.$$

• $p = q\,r, a(q) = y \quad (y \in \mathcal{X})$
C'est un cas particulier du cas suivant, en prenant pour β l'identité.
• $p = q\,r, a(q) = \beta(x_i)_{i\in I} \quad (\beta(x_i)_{i\in I} \in \mathcal{O}(\mathcal{X}^*) \,:\, t)$
Dans ce cas, $\gamma_s((a))(q) = (\gamma_t \circ \beta)(x_i)_{i\in I}$. Vérifions l'égalité :

$$\begin{aligned} \gamma_s((a))[\nu](p) &= (\gamma_t \circ \beta)((\nu(x_i))_{i\in I})(r) \\ &= \gamma(\beta((\nu(x_i))_{i\in I})(r))\,, \\ \gamma_s((a[\nu]))(p) &= \gamma(\beta((\nu(x_i))_{i\in I})(r))\,. \end{aligned}$$

• Cas restant
Les deux valeurs à comparer ne sont pas définies.
⌞

On en déduit immédiatement la proposition suivante.

1.1.8 Proposition (Opérations autorisées : les images par $\mathcal{T}^\infty$)
À toute sorte s et tout morphisme γ de Σ dans Σ associons l'opération unaire $\mathcal{T}^\infty_\gamma \,:\, s$, restriction de $\mathcal{T}^\infty_\gamma$ à $\mathcal{T}^\infty_\Sigma \,:\, s$. L'ensemble des opérations unaires $\mathcal{T}^\infty_\gamma \,:\, s$ est formé d'opérations autorisées.

Démonstration
• Fermeture par composition
Pour tous morphismes γ_1 et γ_2, on a :

$$(\mathcal{T}^\infty_{\gamma_1} \,:\, s) \circ (\mathcal{T}^\infty_{\gamma_2} \,:\, s) = (\mathcal{T}^\infty_{\gamma_1\circ\gamma_2} \,:\, s)\,.$$

• Préservation de la propriété de garde
Sous l'hypothèse de fermeture par composition, on a vu comment prolonger une opération $\mathcal{T}^\infty_\gamma \,:\, s$, en une application de $\mathcal{T}^\infty_\Sigma[\mathcal{X} \cup \mathcal{O}(\mathcal{X})] \,:\, s$ dans lui-même : ce prolongement préserve la propriété de garde.
• Commutativité avec les valuations
Par le lemme précédent, la propriété de commutativité est aussi vérifiée.
⌞

Nous pouvons maintenant résoudre tout système d'équations gardées contenant des opérations autorisées. Étant donné un ensemble d'opérations autorisées $\mathcal{O}$, définissons la fonction Θ de $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$ dans $\mathcal{T}^\infty_\Sigma[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$ réalisant la transformation des membres droits utilisée pour la résolution ; elle est définie inductivement par les équations suivantes, pour tout arbre a appartenant à $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$:
– $a \in \mathcal{T}^\infty_\Sigma[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$:

$$\Theta(a) \stackrel{def}{=} a\,,$$

– $a = f(a_i)_{i\in I}$ $(f \in \mathcal{F})$:

$$\Theta(f(a_i)_{i\in I}) \stackrel{def}{=} f(\Theta(a_i))_{i\in I}\,,$$

– $a = \alpha(a_i)_{i\in I}$ $(\alpha \in \mathcal{O})$:

$$\Theta(\alpha(a_i)_{i\in I}) \stackrel{def}{=} \alpha((\Theta(a_i))_{i\in I})\,.$$

Il est facile de montrer par récurrence structurelle que Θ préserve la sorte : $\Theta(a)$ a la même sorte que a. Comme toute opération autorisée préserve la propriété de garde, il est aussi facile de montrer, toujours par récurrence structurelle, que Θ préserve cette même propriété : si a appartenant à $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$ est gardé, alors $\Theta(a)$ est gardé.
Nous pouvons maintenant énoncer le théorème concernant la résolution des systèmes d'équations avec des opérations.

1.1.9 Théorème (Résolution des systèmes avec opérations)
Soit $\mathcal{O}$ un ensemble d'opérations sur $\mathcal{T}_\Sigma^\infty$ autorisées. Soit

$$(x = a_x)_{x\in\mathcal{X}}$$

un système d'équations récursives, vérifiant pour toute inconnue x les conditions suivantes :
– *le membre droit a_x appartient à $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$ et est gardé,*
– *si x a pour sorte s, alors a_x également.*
Alors ce système admet une unique solution à valeur dans $\mathcal{T}_\Sigma^\infty$.

Démonstration
Associons au système de l'énoncé le système suivant :

$$\begin{array}{l}(x = \Theta(a_x))_{x\in\mathcal{X}}\,,\\ \big(\beta(x_j)_{j\in J} = \beta((\Theta(a_{x_j}))_{j\in J})\big)_{\beta(x_j)_{j\in J}\in\mathcal{O}(\mathcal{X}^*)}\,.\end{array}$$

Les inconnues de ce système sont les variables de l'ensemble $\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)$, les membres droits appartiennent à $\mathcal{T}_\Sigma^\infty[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$, ont même sorte que l'inconnue à laquelle ils sont associés et sont gardés. D'après le théorème 1.1.5 (p. 56), ce système admet donc une unique solution ν, à valeur dans $\mathcal{T}_\Sigma^\infty$. Vérifions que la restriction de ν à $\mathcal{X}$ est solution du système de l'énoncé. Montrons par récurrence structurelle sur a appartenant à $\mathcal{E}_{\Sigma,\mathcal{O}}[\mathcal{X}]$ que

$$\Theta(a)[\nu] = a[\nu]\,.$$

• $a \in \mathcal{T}_\Sigma^\infty[\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)]$
Comme $\Theta(a) = a$, c'est immédiat.
• $a = f(a_i)_{i\in I}$ $(f \in \mathcal{F})$
On a :

$$\begin{aligned}\Theta(f(a_i)_{i\in I})[\nu] &= f(\Theta(a_i))_{i\in I}[\nu] \\ &= f(\Theta(a_i)[\nu])_{i\in I} \\ &= f(a_i[\nu])_{i\in I} \quad \text{(hyp. de réc.)} \\ &= f(a_i)_{i\in I}[\nu]\,.\end{aligned}$$

• $a = \alpha(a_i)_{i\in I}$ $(\alpha \in \mathcal{O})$
On a :

$$\begin{aligned}\Theta(\alpha(a_i)_{i\in I})[\nu] &= \alpha((\Theta(a_i))_{i\in I})[\nu] \\ &= \alpha((\Theta(a_i)[\nu])_{i\in I}) \quad \text{(commut.)} \\ &= \alpha((a_i[\nu])_{i\in I}) \quad \text{(hyp. de réc.)} \\ &= \alpha(a_i)_{i\in I}[\nu]\,.\end{aligned}$$

Il s'ensuit que pour toute inconnue x, on a

$$\nu(x) = a_x[\nu]\,,$$

autrement dit que la restriction de ν à $\mathcal{X}$ est solution du système.
Vérifions l'unicité de la solution. Soit ν' une solution de $(x = a_x)_{x\in\mathcal{X}}$. Montrons que le prolongement de ν' sur $\mathcal{X} \cup \mathcal{O}(\mathcal{X}^*)$ est solution du système associé.
Par la propriété précédente, on a pour toute inconnue x :

$$\nu'(x) = \Theta(a_x)[\nu']\,.$$

Rappelons que le prolongement de ν' sur $\mathcal{O}(\mathcal{X}^*)$ est défini pour toute inconnue $\beta(x_j)_{j\in J}$ de $\mathcal{O}(\mathcal{X}^*)$ par :

$$\nu'(\beta(x_j)_{j\in J}) \stackrel{def}{=} \beta((\nu'(x_j))_{j\in J})\,.$$

On a alors :

$$\begin{aligned}\beta((\Theta(a_{x_j}))_{j\in J})[\nu'] &= \beta((\Theta(a_{x_j})[\nu'])_{j\in J}) \quad \text{(commut.)} \\ &= \beta((a_{x_j}[\nu'])_{j\in J}) \\ &= \beta((\nu'(x_j))_{j\in J})\,.\end{aligned}$$

La valuation ν' est solution du système associé. Par unicité, $\nu' = \nu$: la solution est bien unique.
⌞

Pour conclure, nous disposons d'un moyen simple pour construire des arbres : les systèmes d'équations récursives. Il peut s'agir aussi bien d'arbres étiquetés que d'arbres respectant une signature. Les équations doivent être gardées, et peuvent contenir des opérations sur les arbres, sous certaines conditions.

1.2 Raisonner sur les objets infinis

Nous allons utiliser les systèmes d'équations récursives pour définir des preuves, éventuellement infinies en profondeur. Ces preuves s'entendent relativement à des systèmes d'inférence, ensembles de règles d'inférence. Comme nous l'avons vu en introduction, ce sont les objets infinis qui obligent à étendre les preuves admissibles à celles non fondées, et à ainsi considérer deux *interprétations* d'un système d'inférence : l'*inductive*, admettant les seules preuves bien fondées, et la *co-inductive*, libérant cette contrainte.
Prenons l'exemple des arbres que nous venons d'étudier. L'égalité de deux arbres peut être montrée par l'égalité de leurs racines et l'égalité de leurs sous-arbres. Si les arbres ne sont pas fondés, ce raisonnement ne s'arrête pas : la preuve ne sera pas fondée. Concrètement, l'égalité que nous venons de caractériser correspond à une preuve dans le système d'inférence suivant, défini sur le produit cartésien $(\mathcal{T}_{\mathcal{F}}^{\infty})_{\perp} \times (\mathcal{T}_{\mathcal{F}}^{\infty})_{\perp}$:

$$\text{axiome :} \qquad \frac{\emptyset}{(\perp, \perp)} \quad (\perp : \text{arbre vide}),$$

$$\text{règle d'inférence :} \quad \frac{(a_{/j}, b_{/j})_{j \in \mathcal{P}}}{(a, b)} \quad (a, b \neq \perp, a(\varepsilon) = b(\varepsilon)).$$

Que ce système définisse bien l'égalité est clair : une preuve a la même structure que les deux arbres formant sa conclusion, et la condition d'égalité des racines donne l'égalité des arbres en toute position. Il apparaît donc que si l'arbre est infini en profondeur, la preuve l'est aussi.
Nous introduisons donc les systèmes d'inférence, puis associons à chacun son opérateur d'inférence, qui permet de réaliser une inférence dans le système. Un opérateur d'inférence possède deux points fixes remarquables, le plus petit et le plus grand, que nous caractérisons de deux manières, imprédicative

et itérative. Vient ensuite l'approche déductive, qui vise à caractériser ces points fixes à partir des preuves dans le système d'inférence considéré.
Toute cette étude peut être répétée en considérant une application croissante d'un treillis complet dans lui-même, application représentant l'opérateur d'inférence, comme on le verra.

1.2.1 Prouver dans un système d'inférence

Un *système d'inférence* est un ensemble de règles portant sur des *jugements*, éléments d'un ensemble, noté $\mathcal{U}$. Une *règle* d'inférence sur $\mathcal{U}$ est un couple (Z, z), où $Z \subseteq \mathcal{U}$ est l'ensemble des *prémisses* et $z \in \mathcal{U}$ est la *conclusion*. Une règle est aussi notée

$$\frac{Z}{z}.$$

Il existe une surjection canonique entre les systèmes d'inférence sur $\mathcal{U}$ et les opérateurs croissants sur $\mathcal{U}$[21]. Si Φ est un système d'inférence, alors l'opérateur $\varphi : 2^{\mathcal{U}} \to 2^{\mathcal{U}}$ est défini ainsi :

$$\varphi(Y) = \{z \in \mathcal{U} \mid \exists\, Z \subseteq Y\,.\ (Z, z) \in \Phi\}\,.$$

$\varphi(Y)$ donne ainsi les conclusions que l'on peut déduire de Y en une inférence : φ est ainsi appelé l'*opérateur d'inférence* associé au système Φ. Par exemple, si Y est l'ensemble vide, alors $\varphi(Y)$ est l'ensemble des *axiomes* du système, c'est-à-dire les conclusions des règles sans prémisses.
Réciproquement, étant donné un opérateur φ, le système d'inférence contenant les seules règles (Z, z), avec $z \in \varphi(Z)$, est un antécédent de φ : c'est le plus grand élément, au sens de l'inclusion, de tous les systèmes d'inférence antécédents de φ. On en parlera comme du système d'inférence *associé* à l'opérateur φ.

Considérons un système d'inférence Φ, et l'opérateur associé φ. Par l'intermédiaire de φ, il est possible d'associer des ensembles remarquables au système Φ. En effet, l'opérateur φ possède des points fixes, en particulier un plus petit et un plus grand qui peuvent être caractérisés de manière imprédicative à l'aide des notions de stabilité et de densité. Une partie Z telle que $\varphi(Z) \subseteq Z$ est dite *stable* par l'opérateur φ (et par extension pour le système d'inférence Φ) ; une partie Z telle que $Z \subseteq \varphi(Z)$ est dite *dense* pour φ (ou

[21] Un *opérateur* sur un ensemble E est une application de 2^E dans 2^E (2^E étant l'ensemble des parties de E).

Φ). L'interprétation dans le système d'inférence est la suivante : si les prémisses d'une règle appartiennent à une partie stable Z, alors sa conclusion aussi ; si un jugement appartient à une partie dense, alors il existe une règle de conclusion ce jugement et dont les prémisses appartiennent à la partie dense.
Le théorème suivant montre l'existence de ces points fixes extrémaux, et donne la caractérisation imprédicative annoncée. Il est valide dans tout *treillis complet*, et pas seulement dans l'ensemble des parties.

1.2.1 Théorème (Points fixes (Knaster-Tarski))
Soit $(T, \leq, \vee, \wedge, \bot, \top)$ *un treillis complet*[22]*, et* η *une application croissante de* T *dans* T*. Alors* η *admet un plus petit point fixe* $\mathrm{lfp}(\eta)$ *et un plus grand point fixe* $\mathrm{gfp}(\eta)$*, respectivement le plus petit élément de* T *stable par* η *et le plus grand élément dense pour* η *:*

$$\begin{aligned} \mathrm{lfp}(\eta) &= \min\{x \in T \mid \eta(x) \leq x\}, \\ \mathrm{gfp}(\eta) &= \max\{x \in T \mid x \leq \eta(x)\}. \end{aligned}$$

Démonstration
Notons I_* la borne inférieure des éléments stables par η, $\bigwedge\{x \in T \mid \eta(x) \leq x\}$. On va montrer que I_* est un point fixe. Il s'ensuivra que I_* est stable et est donc le plus petit élément stable par η, et que I_* est le plus petit point fixe de η puisque tout point fixe est stable.
Pour montrer que I_* est stable, on montre que $\eta(I_*)$ est inférieur à tout élément stable x :

$$\begin{aligned} I_* \leq x &\Rightarrow \eta(I_*) \leq \eta(x) \quad (\eta \text{ croissante}) \\ &\Rightarrow \eta(I_*) \leq x \quad (x \text{ stable}). \end{aligned}$$

Par définition de la borne inférieure, $\eta(I_*) \leq I_*$.
On montre maintenant que I_* est dense : comme I_* est stable et η est croissante, $\eta(I_*)$ est stable et $I* \leq \eta(I_*)$, par définition de I_*.
I_*, stable et dense, est bien un point fixe.
Par dualité, on déduit le résultat pour le plus grand point fixe.
⌞

Ces points fixes peuvent aussi être calculés par itération transfinie.

[22] Un *treillis complet* est un ensemble ordonné tel que toute partie Z y admet une borne supérieure, notée $\vee Z$, et une borne inférieure, notée $\wedge Z$. $\vee\emptyset$ est alors le plus petit élément, noté $\bot$, et $\wedge\emptyset$ le plus grand, noté $\top$.

1.2.2 Théorème (Calcul des points fixes)
Soient $(T, \leq, \vee, \wedge, \bot, \top)$ un treillis complet, et η une application croissante de T dans T.
Alors le plus petit point fixe de η peut être calculé par itération transfinie ainsi :

$$\mathrm{lfp}(\eta) = \bigvee_{\alpha} \Delta_\alpha(\eta)\,,$$

où la suite des itérés $(\Delta_\alpha(\eta))_\alpha$, définie sur les ordinaux par l'équation

$$\Delta_\alpha(\eta) = \bigvee_{\beta|\beta<\alpha} \eta(\Delta_\beta(\eta))\,,$$

est croissante et stationnaire à partir d'un certain rang.
De même, le plus grand point fixe de η peut être calculé par itération transfinie ainsi :

$$\mathrm{gfp}(\eta) = \bigwedge_{\alpha} \nabla_\alpha(\eta)\,,$$

où la suite des itérés $(\nabla_\alpha(\eta))_\alpha$, définie sur les ordinaux par l'équation

$$\nabla_\alpha(\eta) = \bigwedge_{\beta|\beta<\alpha} \eta(\nabla_\beta(\eta))\,,$$

est décroissante et stationnaire à partir d'un certain rang.

Démonstration
• Croissance de la suite
On commence par montrer que la suite $(\Delta_\alpha(\eta))_\alpha$ est croissante. Il suffit de montrer par induction transfinie sur α que $\eta(\Delta_\alpha(\eta)) \geq \Delta_\alpha(\eta)$: car on aura alors pour $\beta < \alpha$, $\Delta_\alpha(\eta) \geq \eta(\Delta_\beta(\eta)) \geq \Delta_\beta(\eta)$.
Supposons donc $\forall\, \beta < \alpha\,.\, \eta(\Delta_\beta(\eta)) \geq \Delta_\beta(\eta)$. On a, en utilisant la croissance de η :

$$\begin{aligned}
\eta(\Delta_\alpha(\eta)) &= \eta\Big(\bigvee_{\beta|\beta<\alpha} \eta(\Delta_\beta(\eta))\Big) \quad \text{(déf.)} \\
&\geq \bigvee_{\beta|\beta<\alpha} \eta^2(\Delta_\beta(\eta)) \quad \text{(prop. de } \vee\text{)} \\
&\geq \bigvee_{\beta|\beta<\alpha} \eta(\Delta_\beta(\eta)) \quad \text{(hyp. d'ind.)} \\
&= \Delta_\alpha(\eta)\,.
\end{aligned}$$

De la croissance de la suite $(\Delta_\alpha(\eta))_\alpha$, on déduit que pour tout ordinal α, on a

$$\Delta_{\alpha+1}(\eta) = \eta(\Delta_\alpha(\eta))\,,$$

simplification qui nous sera utile.

• Stationnarité de la suite

Avant de montrer que la suite est stationnaire, montrons d'abord que s'il existe un ordinal λ tel que $\Delta_{\lambda+1}(\eta) = \Delta_\lambda(\eta)$, alors la suite est stationnaire à partir du rang λ. Notons que d'après la remarque précédente, l'hypothèse $\Delta_{\lambda+1}(\eta) = \Delta_\lambda(\eta)$ entraîne que $\eta(\Delta_\lambda(\eta)) = \Delta_\lambda(\eta)$, ce qui signifie que $\Delta_\lambda(\eta)$ est un point fixe de η.

Supposons λ tel que $\Delta_{\lambda+1}(\eta) = \Delta_\lambda(\eta)$. Montrons par induction transfinie que pour tout ordinal $\alpha > \lambda$, alors $\Delta_\alpha(\eta) = \Delta_\lambda(\eta)$.

Soit α un ordinal strictement supérieur à λ et supposons que pour tout ordinal β tel que $\alpha > \beta > \lambda$, on ait $\Delta_\beta(\eta) = \Delta_\lambda(\eta)$. On a alors, en utilisant la croissance de la suite :

$$\begin{aligned}
\Delta_\alpha(\eta) &= \bigvee_{\beta|\beta<\alpha} \eta(\Delta_\beta(\eta)) \\
&= \bigvee_{\beta|\lambda\leq\beta<\alpha} \eta(\Delta_\beta(\eta)) \\
&= \bigvee_{\beta|\lambda\leq\beta<\alpha} \eta(\Delta_\lambda(\eta)) \quad \text{(hyp. d'ind.)} \\
&= \bigvee_{\beta|\lambda\leq\beta<\alpha} \Delta_\lambda(\eta) \\
&= \Delta_\lambda(\eta)\,.
\end{aligned}$$

On montre maintenant que la suite $(\Delta_\alpha(\eta))_\alpha$ est stationnaire. Soit S son support. Inclus dans l'ensemble T, S est un ensemble, de surcroît ordonné par l'ordre induit. Comme la suite est croissante et indexée par la classe des ordinaux, qui est bien ordonnée, S est bien ordonné. Il existe donc un ordinal λ et un unique isomorphisme d'ordre, noté ι, de S vers λ. On va montrer par induction transfinie que pour tout ordinal $\alpha < \lambda$, on a $\iota(\Delta_\alpha(\eta)) = \alpha$. Il s'ensuivra que la suite est stationnaire à partir de λ.

Supposons $\alpha < \lambda$, et pour tout $\beta < \alpha$, $\iota(\Delta_\beta(\eta)) = \beta$. On montre par l'absurde que $\iota(\Delta_\alpha(\eta)) = \alpha$.

Supposons $\iota(\Delta_\alpha(\eta)) < \alpha$. Soit $\beta = \iota(\Delta_\alpha(\eta))$. Par hypothèse d'induction, $\iota(\Delta_\beta(\eta)) = \beta$, puis par injectivité de ι, $\Delta_\beta(\eta) = \Delta_\alpha(\eta)$. Par croissance de la suite, $\Delta_\beta(\eta) = \Delta_{\beta+1}(\eta)$, si bien que la suite est stationnaire à partir de β. Donc α ne peut avoir d'antécédent par ι, ce qui contredit le fait que ι est

un isomorphisme d'ordre.
Supposons $\iota(\Delta_\alpha(\eta)) > \alpha$. Comme la suite est croissante, tout comme ι, α ne peut avoir d'antécédent par ι, ce qui contredit le fait que ι est un isomorphisme d'ordre.
Finalement, $\iota(\Delta_\alpha(\eta)) = \alpha$. Déduisons-en que la suite est stationnaire.
Soit $\alpha = \iota(\Delta_\lambda(\eta))$. Comme $\iota(\Delta_\alpha(\eta)) = \alpha$, on déduit par injectivité de ι que $\Delta_\alpha(\eta) = \Delta_\lambda(\eta)$, puis par croissance de la suite, que $\Delta_\alpha(\eta) = \Delta_{\alpha+1}(\eta)$, ce qui implique que la suite est stationnaire à partir de α, d'où le résultat.
• Calcul du plus petit point fixe
Comme $\Delta_\lambda(\eta) = \Delta_{\lambda+1}(\eta)$, $\Delta_\lambda(\eta)$ est un point fixe de η.
Enfin, montrons que pour tout ordinal α, $\Delta_\alpha(\eta) \leq \mathrm{lfp}(\eta)$. On procède par induction transfinie, et c'est immédiat.
Finalement $\Delta_\lambda(\eta)$ est le plus petit point fixe.
• Cas du plus grand point fixe
Pour le plus grand point fixe, on raisonne par dualité.
⌞

Jusqu'à maintenant, le système d'inférence n'a servi qu'à définir l'opérateur d'inférence associé, et n'a donc pas été utilisé comme système déductif, permettant de prouver des jugements. C'est ce dernier point que nous abordons, en caractérisant par des preuves les plus petit et plus grand points fixes de l'opérateur d'inférence associé.

Considérons un système d'inférence Φ défini sur l'ensemble de jugements $\mathcal{U}$, et définissons ce qu'est une *preuve* dans Φ, ou plus exactement sa représentation par un arbre. Cette définition suppose que pour toute règle d'inférence, l'ensemble de ses prémisses soit décrit sous la forme d'une famille, pas nécessairement injective. On se donne donc un ensemble $\mathcal{P}$ et pour toute règle (Z, z) de Φ une famille $(z_k)_{k\in K}$, avec $K \subseteq \mathcal{P}$, telle que $Z = \{z_k \mid k \in K\}$. Dans ce cas, la règle est notée $((z_k)_{k\in K}, z)$.
Si ces familles ne sont pas précisées, on considère que $\mathcal{P}$ est égal à l'ensemble des jugements, $\mathcal{U}$, et pour toute règle (Z, z) de Φ, on décrit Z par la famille injective $(j)_{j\in Z}$. Dans ce cas, les preuves ainsi construites seront dites *canoniquement associées* au système d'inférence.
Bien souvent, pour un système d'inférence finitaire[23], les prémisses de chaque règle seront indexées par un segment initial des entiers naturels. Lorsque c'est le cas, une règle $((z_k)_{0\leq k\leq n}, z)$ sera notée aussi

$$\frac{z_0 \dots z_n}{z}.$$

[23]Un système d'inférence est *finitaire* lorsque chacune de ses règles a un nombre fini de prémisses.

Pour les arbres de preuves, l'ensemble des positions est engendré par $\mathcal{P}$, et l'ensemble des étiquettes est égal à celui des jugements. Un arbre $a : \mathcal{P}^* \to \mathcal{U}$ représente une preuve s'il vérifie la condition suivante :

> pour toute position p du domaine de a, il existe une règle de Φ, $((z_k)_{k\in K}, z)$, telle que
>
> $$\begin{array}{ll} (i) & a(p) = z\,, \\ (ii) & \{k \in \mathcal{P} \mid p\,k \in \operatorname{dom} a\} = K\,, \\ (iii) & \forall\, k \in K\,.\ a(p\,k) = z_k\,. \end{array}$$

Une alternative intéressante consiste à considérer les jugements comme des sortes, et à prendre pour étiquettes les règles d'inférence, ou mieux, des noms qu'on leur donne ; dans ce cas, le couple des prémisses et de la conclusion d'une règle constitue son profil, et une preuve est alors un arbre respectant la signature concrète ainsi définie. La condition précédente définissant les preuves correspond à la condition d'arité associée à cette signature concrète.

Désormais, la terminologie développée pour les arbres est adoptée pour les preuves, toujours représentées par des arbres : ainsi, par exemple, une sous-preuve est un sous-arbre d'un arbre de preuve, ou une preuve *bien fondée* est un arbre de preuve bien fondé. Quant à la racine $a(\varepsilon)$ d'une preuve a, elle est appelée sa *conclusion*.
L'*interprétation* d'un système d'inférence est définie par la donnée d'un ensemble de preuves, dites *admissibles*. Les conclusions de ces preuves forment les *jugements valides*, autrement dit validés par des preuves admissibles, et correspondent généralement, et dans les cas nous intéressant, à un point fixe de l'opérateur associé au système.
Pour définir des preuves, infinies en profondeur surtout, on utilisera principalement des systèmes quasi-uniformes.

1.2.3 Proposition (Construction récursive de preuves)
Soit Φ un système d'inférence défini sur l'ensemble de jugements $\mathcal{U}$. Soit $(x = a_x)_{x\in\mathcal{X}}$ un système quasi-uniforme défini sur $\mathcal{T}^\infty_{\mathcal{U}}[\mathcal{X}]$, compatible avec le système d'inférence, au sens suivant :
pour toute inconnue x de $\mathcal{X}$,
- *soit a_x est une preuve,*
- *soit $a_x = z(x_k)_{k\in K}$, en notation préfixe, où*
  - *quel que soit k dans K, x_k est une variable,*
  - *$((a_{x_k}(\varepsilon))_{k\in K}, z)$ est une règle de Φ.*

Alors, pour chaque inconnue x, la valeur en x de la solution est une preuve suivant Φ.

Démonstration
Soit ν l'unique solution du système. Considérons la propriété suivante, définie pour tout arbre a de $\mathcal{T}_{\mathcal{U}}^{\infty}$ et toute position p de son domaine, et notée $I(a,p)$:

$$I(a,p) \quad \stackrel{def}{\Leftrightarrow} \quad \exists\,((z_k)_{k\in K}, z) \in \Phi\,.\, \left(\begin{array}{l} a(p) = z \\ \{k \in \mathcal{P} \mid p\,k \in \mathrm{dom}\,a\} = K \\ \forall\, k \in K\,.\; a(p\,k) = z_k \end{array}\right).$$

Vérifions que le lemme des propriétés locales 1.1.4 (p. 54) peut s'appliquer.
Premièrement, le prédicat I définit une propriété locale.
Deuxièmement, montrons que pour toute variable x et toute position p appartenant à $\mathrm{dom}\,a_x$ telle que $a_x(p) \in \mathcal{U}$, on a $I(\nu(x), p)$.
Soit x une variable.
Si a_x est une preuve, alors $\nu(x) = a_x$ et pour toute position p, $I(\nu(x), p)$ est vérifié.
Examinons l'autre cas, où $a_x = z(x_k)_{k\in K}$. On doit vérifier $I(\nu(x), \varepsilon)$.
On a $\nu(x)(\varepsilon) = z$ et pour $k \in \mathcal{P}$, $\nu(x)(k) = a_{x_k}(\varepsilon)$ si $k \in K$ et $\nu(x)(k) = \bot$ sinon. Compte tenu des hypothèses, $I(\nu(x), \varepsilon)$ est bien vérifié, en prenant la règle d'inférence $((a_{x_k}(\varepsilon))_{k\in K}, z)$.
On peut donc conclure que ν a pour invariant I, et donc définit des preuves suivant Φ.
⌞

Les équations définissant des preuves pourront être notées de manière plus explicite ainsi :

$$x = \frac{(x_k)_{k\in K}}{z}.$$

Avant d'aborder la caractérisation des points fixes, et les interprétations inductives et co-inductives, définissons une interprétation mixte, en ajoutant des contraintes pour la formation des preuves. Une contrainte encadre l'utilisation des règles d'inférence : chaque prémisse d'une règle se voit qualifiée de « positive », « négative » ou « neutre », pour indiquer le profil en profondeur de la preuve. Dans la preuve d'un jugement, une règle est utilisable si chacune de ses prémisses, suivant qu'elle est positive, négative ou neutre, est la conclusion d'une sous-preuve respectivement fondée, non fondée ou quelconque. Ainsi, dans une interprétation inductive, toutes les prémisses seront positives, et dans une co-inductive, toutes seront neutres.
On suppose que pour toute règle $((z_k)_{k\in K}, z)$ de Φ, il existe une partition de K en trois ensembles, notés K^+, K^- et K^0 ; les prémisses appartenant à la famille $(z_k)_{k\in K^+}$ sont les prémisses *positives*, celles appartenant à la famille

$(z_k)_{k\in K^-}$ sont les prémisses *négatives*, et enfin celles appartenant à la famille $(z_k)_{k\in K^0}$ sont les prémisses *neutres*. Dans la suite, les règles pourront être notées en précédant les prémisses du signe « + » si elles sont positives, « - » si elles sont négatives et d'aucun signe sinon, comme ceci par exemple pour la règle $((z_k)_{k\in K}, z)$, avec $K = \{0,1,2\}$, $K^+ = \{0\}$, $K^- = \{2\}$ et $K^0 = \{1\}$:

$$\frac{+\ z_0 \quad z_1 \quad -\ z_2}{z}.$$

Une preuve $a : \mathcal{P}^* \to \mathcal{U}$ est admissible dans l'*interprétation mixte* du système d'inférence Φ si elle vérifie

> pour toute position p du domaine de a, il existe une règle de Φ, $((z_k)_{k\in K}, z)$, telle que
>
> (i) $\quad a(p) = z\,,$
> (ii) $\quad \{k \in \mathcal{P} \mid p\,k \in \operatorname{dom} a\} = K\,,$
> (iii) $\quad \forall\, k \in K\,.\ a(p\,k) = z_k\,,$
> (iv) $\quad \forall\, k \in K^+\,.\ a_{/p\,k}$ est bien fondé,
> (v) $\quad \forall\, k \in K^-\,.\ a_{/p\,k}$ n'est pas fondé.

Les conditions (iv) et (v) constituent les contraintes nouvelles sur l'utilisation des règles. Dans une interprétation mixte d'un système d'inférence, on définira généralement les preuves à l'aide de systèmes d'équations récursives. Cependant, la proposition précédente 1.2.3 ne nous permet pas de savoir si une preuve ainsi construite vérifie les contraintes supplémentaires. Raffinons donc cette proposition.

1.2.4 Proposition (Interprétation mixte : Construction récursive de preuves)

Soit Φ un système d'inférence défini sur l'ensemble de jugements $\mathcal{U}$, et considérons une interprétation mixte de ce système.
Soit $(\mathcal{X}^+, \mathcal{X}^-, \mathcal{X}^0)$ une partition de l'ensemble des inconnues $\mathcal{X}$.
Considérons un système quasi-uniforme $(x = a_x)_{x\in\mathcal{X}}$, défini sur $\mathcal{T}^\infty_{\mathcal{U}}[\mathcal{X}]$, compatible avec l'interprétation mixte du système d'inférence, au sens suivant :
pour toute inconnue x de $\mathcal{X}^+$,

- *a_x est une preuve bien fondée admissible dans l'interprétation mixte,*

pour toute inconnue x de $\mathcal{X}^-$,

- *soit a_x est une preuve non fondée admissible dans l'interprétation mixte,*
- *soit $a_x = z(x_k)_{k\in K}$, en notation préfixe, où*

- $((a_{x_k}(\varepsilon))_{k\in K}, z)$ *est une règle de* Φ,
- K^- *n'est pas vide,*
- *quel que soit* k *dans* K^+, x_k *appartient à* $\mathcal{X}^+$,
- *quel que soit* k *dans* K^-, x_k *appartient à* $\mathcal{X}^-$,
- *quel que soit* k *dans* K^0, x_k *appartient à* $\mathcal{X}$,

pour toute inconnue x *de* $\mathcal{X}^0$,

- *soit* a_x *est une preuve admissible dans l'interprétation mixte,*
- *soit* $a_x = z(x_k)_{k\in K}$, *en notation préfixe, où*
  - $((a_{x_k}(\varepsilon))_{k\in K}, z)$ *est une règle de* Φ,
  - *quel que soit* k *dans* K^+, x_k *appartient à* $\mathcal{X}^+$,
  - *quel que soit* k *dans* K^-, x_k *appartient à* $\mathcal{X}^-$,
  - *quel que soit* k *dans* K^0, x_k *appartient à* $\mathcal{X}$.

Alors, pour chaque inconnue x, *la valeur en* x *de la solution est une preuve admissible dans l'interprétation mixte de* Φ.

La démonstration est analogue à celle de la proposition précédente 1.2.3.

Démonstration
Soit ν l'unique solution du système. Considérons la propriété suivante, définie pour tout arbre a de $\mathcal{T}^\infty_{\mathcal{U}}$ et toute position p de son domaine, et notée $I(a,p)$:

$$I(a,p) \quad \stackrel{def}{\Leftrightarrow} \quad \exists\,((z_k)_{k\in K}, z) \in \Phi\,.\, \left(\begin{array}{l} a(p) = z \\ \{k \in \mathcal{P} \mid p\,k \in \operatorname{dom} a\} = K \\ \forall\, k \in K\,.\; a(p\,k) = z_k \\ \forall\, k \in K^+\,.\; a_{/p\,k} \text{ est bien fondé} \\ \forall\, k \in K^-\,.\; a_{/p\,k} \text{ n'est pas fondé} \end{array}\right).$$

Vérifions que le lemme des propriétés locales 1.1.4 (p. 54) peut s'appliquer.
Premièrement, le prédicat I définit bien une propriété locale.
Deuxièmement, montrons que pour toute variable x et toute position p appartenant à $\operatorname{dom} a_x$ telle que $a_x(p) \in \mathcal{U}$, on a $I(\nu(x), p)$.
Soit x une variable.
Si a_x est une preuve admissible, alors $\nu(x) = a_x$ et pour toute position p, $I(\nu(x), p)$ est vérifié.
Examinons l'autre cas, où $a_x = z(x_k)_{k\in K}$. On doit vérifier $I(\nu(x), \varepsilon)$.
On a $\nu(x)(\varepsilon) = z$ et pour $k \in \mathcal{P}$, $\nu(x)(k) = a_{x_k}(\varepsilon)$ si $k \in K$ et $\nu(x)(k) = \bot$ sinon.
Compte tenu des hypothèses, les trois premières conditions sont vérifiées, en prenant la règle d'inférence $((a_{x_k}(\varepsilon))_{k\in K}, z)$.
Si $k \in K^+$, on a $\nu(x)_{/k} = a_{x_k}$, et $\nu(x)_{/k}$ est bien fondé.

Supposons enfin $k \in K^-$. On cherche à montrer que $\nu(x)_{/k}$ n'est pas fondé. Comme $\nu(x)_{/k} = \nu(x_k)$, avec $x_k \in \mathcal{X}^-$, il suffit de montrer que pour tout $x \in \mathcal{X}^-$, il existe une suite de preuves $(A^x_n)_n$ décroissant suivant $\prec$ (la relation « est un sous-arbre de ») commençant en $\nu(x)$. On construit cette famille de suites par récurrence en imposant que pour tout n, ou bien A^x_n est une preuve admissible non fondée, ou bien il existe $y \in \mathcal{X}^-$ tel que $A^x_n = \nu(y)$.

- $n = 0$

Pour tout $x \in \mathcal{X}^-$, on pose $A^x_0 = \nu(x)$.

- $n > 0$

Supposons que pour tout $x \in \mathcal{X}^-$, A^x_{n-1} est défini.
Soit $x \in \mathcal{X}^-$.
Si A^x_{n-1} est une preuve admissible non fondée, il existe au moins une sous-preuve immédiate, admissible et non fondée, et on en choisit une pour A^x_n.
Supposons $A^x_{n-1} = \nu(y)$, où $y \in \mathcal{X}^-$. Si a_y est une preuve admissible non fondée, on a $\nu(y) = a_y$, et on est ramené au cas précédent.
Supposons donc $a_y = z(x_k)_{k\in K}$. Par hypothèse, il existe $k \in K^-$ vérifiant $x_k \in \mathcal{X}^-$ et on pose $A^x_n = \nu(x_k)$.
On peut donc conclure que ν a pour invariant I, et donc définit des preuves admissibles dans l'interprétation mixte de Φ.
⌞

Comme précédemment, les équations définissant des preuves admissibles pourront être notées de manière plus explicite en précisant les contraintes, comme dans cet exemple :

$$x = \frac{+\ x_0 \quad x_1 \quad -\ x_2}{z}.$$

Bien que nous utilisions à plusieurs reprises l'interprétation mixte de systèmes d'inférence dans les chapitres suivants, nous ne poursuivons pas son étude, car elle ne donne pas lieu à une caractérisation intéressante par points fixes. Revenons donc aux points fixes extrémaux, et à leur caractérisation par des preuves.

Précisément, quelle notion d'admissibilité pour les preuves correspond aux plus petit et plus grand points fixes de l'opérateur d'inférence ? Le théorème suivant apporte la réponse.

1.2.5 Définition et théorème (Interprétation inductive et co-inductive d'un système d'inférence)

Soit Φ *un système d'inférence.*
Dans son interprétation inductive*, où les preuves admissibles sont les preuves*

bien fondées, l'ensemble des jugements valides est égal au plus petit point fixe de l'opérateur d'inférence associé.
Dans son interprétation co-inductive*, où toutes les preuves sont admissibles, l'ensemble des jugements valides est égal au plus grand point fixe de l'opérateur d'inférence associé.*

Démonstration
Soit φ l'opérateur associé à Φ. Soient $\mathcal{P}_\Delta(\Phi)$ l'ensemble des preuves bien fondées et $\mathcal{P}_\nabla(\Phi)$ l'ensemble de toutes les preuves de Φ. Soient enfin $\Delta(\Phi)$ et $\nabla(\Phi)$ les ensembles de jugements validés par les preuves de $\mathcal{P}_\Delta(\Phi)$ et de $\mathcal{P}_\nabla(\Phi)$ respectivement.
• $\mathrm{lfp}(\varphi) = \Delta(\Phi)$
◦ $\mathrm{lfp}(\varphi) \subseteq \Delta(\Phi)$
Il suffit de montrer que $\varphi(\Delta(\Phi)) \subseteq \Delta(\Phi)$.
Soit $z \in \varphi(\Delta(\Phi))$.
Il existe $Z \subseteq \Delta(\Phi)$ tel que $(Z, z) \in \Phi$. Supposons que Z soit décrit par la famille $(z_k)_{k\in K}$. Comme $Z \subseteq \Delta(\Phi)$, on peut associer à chaque k de K une preuve bien fondée $a_k \in \mathcal{P}_\Delta(\Phi)$ ayant z_k comme conclusion. Considérons le système suivant, d'inconnues $(x_\varepsilon, (x_k)_{k\in K})$:

$$\begin{aligned} x_\varepsilon &= \frac{(x_k)_{k\in K}}{z}, \\ x_k &= a_k \quad (k \in K)\,. \end{aligned}$$

Il est quasi-uniforme et compatible avec Φ : d'après la proposition 1.2.3 (p. 73), la valeur de l'unique solution en x_ε est une preuve de conclusion z, de plus bien fondée. Ainsi, z appartient ainsi à $\Delta(\Phi)$. Finalement, $\Delta(\Phi)$ est stable, ce qui montre la première inclusion.
◦ $\Delta(\Phi) \subseteq \mathrm{lfp}(\varphi)$
Nous procédons par récurrence structurelle sur l'ensemble des preuves bien fondées.
Soit a une telle preuve, et supposons $\forall\, a' \prec a\,.\ a'(\varepsilon) \in \mathrm{lfp}(\varphi)$.
Comme l'ensemble des prémisses de la conclusion de a, $\{a(k) \mid k \in \mathcal{P} \cap \mathrm{dom}\, a\}$, est égal à $\{a'(\varepsilon) \mid a' \prec a\}$ et est donc inclus par l'hypothèse de récurrence dans $\mathrm{lfp}(\varphi)$, on déduit par la stabilité de $\mathrm{lfp}(\varphi)$ que $a(\varepsilon) \in \mathrm{lfp}(\varphi)$.
Finalement, $\forall\, a \in \mathcal{P}_\Delta(\Phi)\,.\ a(\varepsilon) \in \mathrm{lfp}(\varphi)$.
• $\mathrm{gfp}(\varphi) = \nabla(\Phi)$
◦ $\mathrm{gfp}(\varphi) \subseteq \nabla(\Phi)$
Comme $\mathrm{gfp}(\varphi)$ est dense, on peut associer à tout $z \in \mathrm{gfp}(\varphi)$ une règle $((j(z,k))_{k\in K_z}, z)$ de Φ, avec $\{j(z,k) \mid k \in K_z\} \subseteq \mathrm{gfp}(\varphi)$. Considérons le

système suivant, d'inconnues $(x_z)_{z\in\mathrm{gfp}(\varphi)}$:

$$\left(x_z = \frac{(x_{j(z,k)})_{k\in K_z}}{z}\right)_{z\in\mathrm{gfp}(\varphi)}.$$

Il est quasi-uniforme et compatible avec Φ : d'après la proposition 1.2.3 (p. 73), pour tout z de $\mathrm{gfp}(\varphi)$, la valeur de l'unique solution en x_z est une preuve de conclusion z.
◦ $\nabla(\Phi) \subseteq \mathrm{gfp}(\varphi)$
Montrons que $\nabla(\Phi) \subseteq \varphi(\nabla(\Phi))$, ce qui permettra de conclure.
Soit $z \in \nabla(\Phi)$. Il existe $a \in \mathcal{P}_\nabla(\Phi)$ tel que $a(\varepsilon) = z$. Soit $Z = \{a(k) \mid k \in \mathcal{P} \cap \mathrm{dom}\, a\}$. On a $Z \subseteq \nabla(\Phi)$ et $(Z, z) \in \Phi$. Ainsi $z \in \varphi(\nabla(\Phi))$. L'ensemble $\nabla(\Phi)$ est donc dense, d'où $\nabla(\Phi) \subseteq \mathrm{gfp}(\varphi)$.
⌞

Comme corollaire de ce théorème, il vient que la validité d'un jugement dans un système d'inférence ne dépend pas de la forme retenue pour les preuves, précisément de la manière dont les prémisses des règles sont décrites.
Ce théorème justifie aussi un changement de terminologie. Si Φ est un système d'inférence, d'opérateur associé φ, alors le plus petit point fixe de φ est appelé l'ensemble *engendré inductivement* par Φ, noté $\Delta(\Phi)$, le qualificatif « inductif » signifiant la bonne fondation des preuves admissibles ; quant au plus grand point fixe de φ, il est appelé l'ensemble *engendré co-inductivement* par Φ, noté $\nabla(\Phi)$, le qualificatif « co-inductif » signifiant que toutes les preuves sont admissibles.

Avant de poursuivre notre étude, développons un exemple particulièrement utile d'interprétation co-inductive.

1.2.2 Comparer des arbres

Nous illustrons la démarche déductive en poursuivant et détaillant un exemple déjà abordé, et qui nous sera utile par la suite, la comparaison entre termes (soit entre arbres respectant une signature).

Dans un terme, une valeur de non-définition $\bot_s$ (s étant une sorte) apparaît nécessairement en une feuille. On peut compléter un terme comportant des valeurs de non-définition en les remplaçant par des termes non triviaux, pour obtenir un terme mieux défini. L'ordre associé à cette opération de remplacement a été décrit précédemment comme la relation « est moins défini que » ; il peut être compris aussi comme la relation « est un sous-terme initial », en interprétant les valeurs de non-définition comme des arbres vides

(pour chaque sorte), ce que nous avons déjà fait ; on choisit plutôt de le nommer « ordre d'approximation » : étant donné un terme, on peut chercher à l'approcher en utilisant des termes moins définis que lui. Nous allons définir cette relation de deux manières :

- par similarité, à partir de l'interprétation co-inductive d'un système d'inférence,
- par compatibilité, en étendant aux termes, considérés comme des applications de l'ensemble des positions dans l'ensemble des étiquettes augmenté des valeurs de non-définition, la relation d'ordre naturelle sur l'ensemble d'arrivée.

Dans la suite, on considère la signature pointée $\Sigma_\perp$, définie sur l'ensemble de sortes $\mathcal{S}$, l'ensemble de positions $\mathcal{P}^*$ et l'ensemble d'étiquettes $\mathcal{F}$.
Commençons par la définition de l'extension. L'ensemble $\mathcal{F} \cup \{\perp_s \mid s \in \mathcal{S}\}$ peut être muni d'une relation d'ordre strict $<$ définie pour toute sorte s et toute étiquette f de sorte s par :

$$\perp_s < f\,.$$

L'*ordre d'approximation* est l'extension à $\mathcal{T}^\infty_{\Sigma_\perp}$ compatible avec la relation précédente, et se définit ainsi, pour tout terme a et b de $\mathcal{T}^\infty_{\Sigma_\perp}$:

$$a \leq b \overset{def}{\Leftrightarrow} (\operatorname{dom} a \subseteq \operatorname{dom} b) \wedge (\forall\, p \in \operatorname{dom} a\,.\ a(p) \leq b(p))\,.$$

Il s'agit évidemment d'une relation d'ordre.
Voyons maintenant la définition par similarité de cette même relation. Considérons le système d'inférence suivant, noté $\Phi_\leq$, défini sur le produit cartésien $\mathcal{T}^\infty_{\Sigma_\perp} \times \mathcal{T}^\infty_{\Sigma_\perp}$ par les règles de la forme suivante :

$$\frac{\emptyset}{(\perp_s, a)} \quad (a \in \mathcal{T}^\infty_{\Sigma_\perp} \,:\, s)\,,$$

$$\frac{\big((\sigma_1(y), \sigma_2(y))\big)_{y \in \operatorname{var}(e)}}{(e[\sigma_1], e[\sigma_2])} \quad \left(\begin{array}{l} e \in \mathcal{T}^\infty_{\Sigma_\perp}[\mathcal{X}], e(\varepsilon) \in \mathcal{F} \\ \sigma_1, \sigma_2 : \operatorname{var}(e) \xrightarrow{\text{v}} \mathcal{T}^\infty_{\Sigma_\perp} \end{array} \right)\,.$$

Nous allons montrer que l'ensemble engendré co-inductivement par $\Phi_\leq$ est égal à la relation d'ordre d'approximation. Dans la démonstration que nous donnons, nous utilisons une hypothèse portant sur la taille de l'ensemble des variables $\mathcal{X}$: précisément, on suppose que pour chaque sorte s, il est possible de plonger l'ensemble des positions $\mathcal{P}^*$ dans l'ensemble des variables de sorte s, $\mathcal{X}_s$, plongement qu'on décrit par la famille $(x^s_p)_{p \in \mathcal{P}^*}$.

1.2.6 Proposition (Définition par similarité de l'ordre d'approximation)

L'ensemble engendré co-inductivement par $\Phi_{\leq}$ est égal à la relation d'ordre d'approximation : pour tous termes a et b de $\mathcal{T}^{\infty}_{\Sigma_\perp}$, on a

$$(a,b) \in \nabla(\Phi_{\leq}) \Leftrightarrow a \leq b\,.$$

Démonstration

• $a \leq b \Rightarrow (a,b) \in \nabla(\Phi_{\leq})$

Supposons $a \leq b$. On associe tout d'abord à a un terme e, appartenant à $\mathcal{T}^{\infty}_{\Sigma}[\mathcal{X}]$ et obtenu en remplaçant les valeurs de non-définition par des variables, de la manière suivante, pour toute position p de $\operatorname{dom} a$:

$$e(p) = \begin{cases} a(p) & \text{si } a(p) \in \mathcal{F}\,, \\ x^s_p & \text{si } a(p) = \perp_s\,. \end{cases}$$

Considérons la valuation $\sigma_1 : \operatorname{var}(e) \xrightarrow{\text{v}} \mathcal{T}^{\infty}_{\Sigma_\perp}$ définie par :

$$\sigma_1(x^s_p) = \perp_s\,.$$

On a évidemment $e[\sigma_1] = a$.

Considérons maintenant l'application $\sigma_2 : \operatorname{var}(e) \to \mathcal{T}^{\infty}_{\Sigma_\perp}$ définie par :

$$\sigma_2(x^s_p) = b_{/p}\,.$$

Elle respecte les sortes, puisque si x^s_p est une variable de e, alors $a(p)$, égal à $\perp_s$, a pour sorte s, et donc $b(p)$ aussi : σ_2 est donc une valuation.

Montrons que $e[\sigma_2] = b$.

Soit p une position. On a :

$$e[\sigma_2](p) = \begin{cases} e(p) & \text{si } e(p) \in \mathcal{F}\,, \\ \sigma_2(x^s_q)(r) & \text{si } p = q\,r, e(q) = x^s_q\,, \\ \perp & \text{sinon.} \end{cases}$$

Dans le premier cas, on a $e(p) = a(p) = b(p)$, et dans le second, $\sigma_2(x^s_q)(r) = b(p)$. Finalement, pour toute position $p \in \operatorname{dom} e[\sigma_2]$, $e[\sigma_2](p) = b(p)$, d'où par le lemme 1.1.3 (p. 52), l'égalité $e[\sigma_2] = b$.

Pour conclure, définissons un système d'équations récursives, d'inconnues $(x, (x_y)_{y \in \operatorname{var}(e)})$, par les équations suivantes :

$$\begin{aligned} x &= \frac{(x_y)_{y \in \operatorname{var}(e)}}{(e[\sigma_1], e[\sigma_2])}\,, \\ x_y &= \frac{\emptyset}{(\sigma_1(y), \sigma_2(y))} \quad (y \in \operatorname{var}(e))\,. \end{aligned}$$

Ce système quasi-uniforme est compatible avec $\Phi_{\leq}$; d'après la proposition 1.2.3 (p. 73), la valeur de la solution en x est une preuve de (a, b) dans $\Phi_{\leq}$.
• $(a, b) \in \nabla(\Phi_{\leq}) \Rightarrow a \leq b$
On montre par récurrence sur l'ensemble des positions, muni de la relation bien fondée $\prec_{\mathrm{s}}^{+}$ (la relation suffixe), que pour toute position p et pour tout couple (a, b) de $\nabla(\Phi_{\leq})$, on a

$$p \in \operatorname{dom} a \Rightarrow p \in \operatorname{dom} b \wedge a(p) \leq b(p)\,.$$

On déduit de cette dernière implication évidemment que $a \leq b$. Remarquons aussi que si $a = \bot_s$, alors $\operatorname{dom} a = \varepsilon$, et l'implication est vérifiée pour toute position p.
Soit p une position et supposons que pour toute position r telle que $r \prec_{\mathrm{s}}^{+} p$, la propriété est vérifiée.
Soit (a, b) appartenant à $\nabla(\Phi_{\leq})$, et considérons le seul cas intéressant, où il existe $e \in \mathcal{T}^{\infty}_{\Sigma_\bot}$ et $\sigma_1, \sigma_2 : \operatorname{var}(e) \to \mathcal{T}^{\infty}_{\Sigma_\bot}$ tels que $a = e[\sigma_1]$, $b = e[\sigma_2]$, $e(\varepsilon) \in \mathcal{F}$ et $\forall\, y \in \operatorname{var}(e)\,.\ (\sigma_1(y), \sigma_2(y)) \in \overline{\nabla}(\Phi_{\leq})$.
Distinguons deux cas.
• $p = \varepsilon$
Dans ce cas, p appartient à $\operatorname{dom} a$ et $\operatorname{dom} b$ et on a $a(p) = e(p) = b(p)$.
• $p \neq \varepsilon$
Supposons $p \in \operatorname{dom} a$. Deux cas sont possibles.
◦ $e(p) \in \mathcal{F} \cup \{\bot_s \mid s \in \mathcal{S}\}$
Alors $a(p) = e(p) = b(p)$.
◦ $p = q\,r, e(q) = x (x \in \mathcal{X})$
Alors $a(p) = \sigma_1(x)(r)$ et $b(p) = \sigma_2(x)(r)$. Comme $e(\varepsilon) \in \mathcal{F}$, $r \prec_{\mathrm{s}}^{+} p$, et on peut appliquer l'hypothèse de récurrence à r et au couple $(\sigma_1(x), \sigma_2(x))$ de $\nabla(\Phi_{\leq})$ pour conclure.
⌞

Après la définition par similarité de la relation d'ordre, il est possible d'en donner une de l'égalité, avec des règles très voisines. Soit $\Phi_{=}$ le système d'inférence suivant, défini sur le produit cartésien $\mathcal{T}^{\infty}_{\Sigma_\bot} \times \mathcal{T}^{\infty}_{\Sigma_\bot}$ par les règles suivantes :

$$\frac{\emptyset}{(\bot_s, \bot_s)} \quad (s \in \bot)\,,$$

$$\frac{(\sigma_1(y), \sigma_2(y))_{y \in \operatorname{var}(e)}}{(e[\sigma_1], e[\sigma_2])} \quad \left(\begin{array}{l} e \in \mathcal{T}^{\infty}_{\Sigma_\bot}, e(\varepsilon) \in \mathcal{F}, \\ \sigma, \sigma' : \operatorname{var}(e) \to \mathcal{T}^{\infty}_{\Sigma_\bot} \end{array} \right).$$

Ce système définit bien l'égalité de deux termes, égalité que nous avons définie précédemment sous la forme d'une extension compatible aux termes de l'égalité entre étiquettes et valeurs de non-définition (cf. p. 52).

1.2.7 Proposition (Définition par bisimilarité de l'égalité)
L'ensemble engendré co-inductivement par $\Phi_=$ est égal à la relation d'égalité entre termes : pour tous termes a et b de $\mathcal{T}^\infty_{\Sigma_\perp}$, on a

$$(a, b) \in \nabla(\Phi_=) \Leftrightarrow a = b\,.$$

Démonstration
• $(a, b) \in \nabla(\Phi_=) \Rightarrow a = b$
Si $(a, b) \in \nabla(\Phi_=)$, alors $(a, b) \in \nabla(\Phi_\leq)$ et par symétrie, $(b, a) \in \nabla(\Phi_\leq)$. Par la proposition précédente, $a \leq b$ et $b \leq a$, soit $a = b$.
• $a = b \Rightarrow (a, b) \in \nabla(\Phi_=)$
On vérifie facilement que pour tout terme a de $\mathcal{T}^\infty_{\Sigma_\perp}$, le couple (a, a) est un axiome de $\Phi_=$.
⌞

1.2.3 Prouver dans un treillis complet

Résumons-nous : on a associé à un système d'inférence sur $\mathcal{U}$ un opérateur croissant défini sur $\mathcal{U}$ et caractérisé son plus petit point fixe et son plus grand. Le plus petit point fixe est, au choix,
- le plus petit ensemble stable,
- la limite d'une suite transfinie croissante d'itérations partant de l'ensemble vide,
- l'ensemble des jugements validés par des preuves bien fondées,

et son plus grand point fixe :
- le plus grand ensemble dense,
- la limite d'une suite transfinie décroissante d'itérations partant de l'ensemble de tous les jugements,
- l'ensemble des jugements validés par des preuves.

Les deux premières caractérisations ont été montrées en utilisant seulement le fait que l'ensemble des parties d'un ensemble est un treillis complet. Une question se pose : peut-on caractériser les points fixes d'une application croissante sur un treillis complet quelconque en termes de preuves ? Si oui, dans quel système d'inférence ? La réponse est affirmative, en voici le développement.

Considérons un treillis complet $(\mathcal{U}, \leq, \vee, \wedge, \perp, \top)$, et une application croissante η de $\mathcal{U}$ dans $\mathcal{U}$. On va transformer η en un opérateur sur $\mathcal{U}$, de manière

à ensuite construire un système d'inférence.
La transformation la plus simple, qui consiste à associer à toute partie l'ensemble formé des images de ses éléments, ne convient pas. En effet, son plus petit point fixe est l'ensemble vide.
En revanche, il est possible de représenter le treillis complet $\mathcal{U}$ à l'intérieur de l'ensemble des parties de $\mathcal{U}$, $2^{\mathcal{U}}$, en utilisant les *idéaux d'ordre principaux*[24]. C'est qu'en effet tout ensemble ordonné est isomorphe à l'ensemble de ses idéaux principaux, ordonné par l'inclusion. Soit ι l'isomorphisme d'ordre entre $\mathcal{U}$ et l'ensemble de ses idéaux principaux, $\leq\mathcal{U}$, défini par :

$$\iota(z) \stackrel{def}{=} \leq z\,,$$

où $\leq z$ est l'idéal principal engendré par z, soit

$$\leq z \stackrel{def}{=} \{y \in \mathcal{U} \mid y \leq z\}\,.$$

Cette correspondance entre les ensembles ordonnés et les ensembles d'idéaux principaux peut être étendue pour former un foncteur entre les deux catégories suivantes, qui sont équivalentes :

- objets : tout ensemble ordonné,
  morphismes : toute application croissante,
- objets : tout ensemble formé des idéaux principaux d'un ensemble ordonné,
  morphismes : toute application croissante (les objets étant ordonnés par l'inclusion).

Ce foncteur transforme ainsi η en $\iota \circ \eta \circ \iota^{-1}$, de $\leq\mathcal{U}$ dans lui-même.
Il reste à établir la relation entre l'ensemble des parties de $\mathcal{U}$ et $\leq\mathcal{U}$, grâce à la complétude de $\mathcal{U}$. Soit donc $(\pi_* : 2^{\mathcal{U}} \to \leq\mathcal{U}, \pi^* : \leq\mathcal{U} \to 2^{\mathcal{U}})$ la connexion de Galois[25] définie par :

$$\begin{aligned} \pi_*(Z) &\stackrel{def}{=} \leq(\vee Z)\,, \\ \pi^*(Y) &\stackrel{def}{=} Y\,. \end{aligned}$$

[24] Un *idéal d'ordre* est une section initiale dirigée d'un ensemble ordonné ; une *section initiale* est une partie stable par minoration, et une partie est *dirigée* si elle peut être munie d'une loi de composition interne associant à tout couple un de leurs majorants. Un idéal est *principal* s'il est engendré par un élément : c'est donc la section initiale fermée engendrée par cet élément.

[25] Étant donné deux ensembles ordonnés $(A, \leq)$ et $(C, \leq)$, une *connexion de Galois* entre A et C est un couple d'applications $(\pi_* : A \to C, \pi^* : C \to A)$ vérifiant pour tout a de A et tout c de C, $a \leq \pi^*(c) \Leftrightarrow \pi_*(a) \leq c$.

À η, on peut maintenant associer un opérateur sur $\mathcal{U}$, φ, défini par

$$\varphi \stackrel{def}{=} \pi^* \circ \iota \circ \eta \circ \iota^{-1} \circ \pi_* \,.$$

Il est facile de vérifier que z est un point fixe de η si et seulement si $\leq z$ est un point fixe de φ. On peut donc construire le système d'inférence à partir de φ, ce qui conduit au théorème suivant.

1.2.8 Théorème (Treillis complet : interprétation inductive et co-inductive) pour points fixes

Soit $(\mathcal{U}, \leq, \vee, \wedge, \bot, \top)$ *un treillis complet. Soit* η *une application croissante de* $\mathcal{U}$ *dans* $\mathcal{U}$*.*
Considérons le système d'inférence Φ *formé des règles de la forme :*

$$\frac{Z}{z} \quad (Z \subseteq \mathcal{U}, z \leq \eta(\vee Z)) \,.$$

Alors :

- *l'ensemble engendré inductivement par* Φ *est l'idéal principal engendré par le plus petit point fixe de* η *:*

$$\Delta(\Phi) \;=\; \leq \mathrm{lfp}(\eta) \,,$$

- *l'ensemble engendré co-inductivement par* Φ *est l'idéal principal engendré par le plus grand point fixe de* η *:*

$$\nabla(\Phi) \;=\; \leq \mathrm{gfp}(\eta) \,.$$

Démonstration
On reprend les notations d'avant l'énoncé.
Pour une partie Z de $\mathcal{U}$, évaluons $\varphi(Z)$; on obtient $\leq \eta(\vee Z)$. Il s´ensuit qu'un point fixe de φ est nécessairement un idéal principal. De plus, pour tout jugement z, on a :

$$\eta(z) = z \Leftrightarrow \varphi(\leq z) = \leq z \,.$$

Les points fixes de η sont donc transformés par ι en ceux de φ, et réciproquement les points fixes de φ sont transformés par ι^{-1} en ceux de η.
Soit enfin φ' l'opérateur associé à Φ. On a :

$$\begin{aligned}
\varphi'(Y) &= \{z \in \mathcal{U} \mid \exists\, Z \subseteq \mathcal{U} \,.\; (Z, z) \in \Phi \wedge Z \subseteq Y\} \\
&= \{z \in \mathcal{U} \mid \exists\, Z \subseteq \mathcal{U} \,.\; z \leq \eta(\vee Z) \wedge Z \subseteq Y\} \\
&= \{z \in \mathcal{U} \mid z \leq \eta(\vee Y)\} \\
&= \leq \eta(\vee Y) \\
&= \varphi(Y) \,.
\end{aligned}$$

L'opérateur associé à Φ est bien φ. On peut donc conclure par le théorème 1.2.5 (p. 77) et la correspondance entre les points fixes de φ et ceux de η.
⌞

Chaque règle (Z, z) du système d'inférence Φ peut se décomposer en une règle liée à η,

$$\frac{Z}{\eta(\vee Z)},$$

et une règle d'affaiblissement, liée à l'ordre $\leq$,

$$\frac{\{\eta(\vee Z)\}}{z},$$

instance du schéma de règle suivant, où $z \leq y$,

$$\frac{\{y\}}{z}.$$

Si seules les preuves des points fixes de η importent, il est possible de se passer des règles d'affaiblissement : les idéaux principaux ne sont alors plus prouvés dans leur entier.

1.2.9 Théorème (Treillis complet (bis) : interprétation inductive et co-inductive) pour points fixes

Soit $(\mathcal{U}, \leq, \vee, \wedge, \bot, \top)$ un treillis complet. Soit η une application croissante de $\mathcal{U}$ dans lui-même.
Considérons le système d'inférence Φ' formé des règles de la forme :

$$\frac{Z}{\eta(\vee Z)} \quad (Z \subseteq \mathcal{U}).$$

Alors :

- *l'ensemble engendré inductivement par Φ' admet un plus grand élément, le plus petit point fixe de η :*

$$\max \Delta(\Phi') \quad = \quad \mathrm{lfp}(\eta),$$

- *l'ensemble engendré co-inductivement par Φ' admet un plus grand élément, le plus grand point fixe de η :*

$$\max \nabla(\Phi') \quad = \quad \mathrm{gfp}(\eta).$$

Démonstration

• $\max \Delta(\Phi') = \mathrm{lfp}(\eta)$

Il suffit de montrer que $\vee\Delta(\Phi') = \mathrm{lfp}(\eta)$. En effet, dans ce cas, considérons la règle de Φ'

$$\frac{\Delta(\Phi')}{\eta(\vee\Delta(\Phi'))}.$$

$\eta(\vee\Delta(\Phi'))$ admet donc une preuve bien fondée et appartient ainsi à $\Delta(\Phi')$; par hypothèse, c'est justement $\vee\Delta(\Phi')$, qui est donc le plus grand élément de $\Delta(\Phi')$.

◦ $\vee\Delta(\Phi') \leq \mathrm{lfp}(\eta)$

Comme $\Delta(\Phi') \subseteq \Delta(\Phi)$, on déduit du théorème précédent l'inégalité.

◦ $\mathrm{lfp}(\eta) \leq \vee\Delta(\Phi')$

Comme précédemment, de la règle

$$\frac{\Delta(\Phi')}{\eta(\vee\Delta(\Phi'))},$$

on déduit que $\eta(\vee\Delta(\Phi'))$ appartient à $\Delta(\Phi')$, et est donc inférieur ou égal à $\vee\Delta(\Phi')$: $\vee\Delta(\Phi')$ est ainsi stable par η. Le théorème de Tarski (1.2.1 (p. 69)) permet de conclure.

• $\max \nabla(\Phi') = \mathrm{gfp}(\eta)$

On montre tout d'abord que $\vee\nabla(\Phi') \leq \mathrm{gfp}(\eta)$, puis que $\mathrm{gfp}(\eta)$ appartient à $\nabla(\Phi')$. On en déduit évidemment que $\mathrm{gfp}(\eta)$ est le maximum de $\nabla(\Phi')$.

◦ $\vee\nabla(\Phi') \leq \mathrm{gfp}(\eta)$

Comme $\Delta(\Phi') \subseteq \Delta(\Phi)$, on déduit du théorème précédent l'inégalité.

◦ $\mathrm{gfp}(\eta) \in \nabla(\Phi')$

Considérons l'équation récursive suivante :

$$x = \frac{x}{\mathrm{gfp}(\eta)}.$$

Comme le système d'inférence Φ' contient la règle d'inférence

$$\frac{\mathrm{gfp}(\eta)}{\mathrm{gfp}(\eta)},$$

cette équation forme un système quasi-uniforme compatible avec Φ' ; d'après la proposition 1.2.3 (p. 73), sa solution est donc une preuve de $\mathrm{gfp}(\eta)$, qui appartient ainsi à $\nabla(\Phi')$.

⌞

1.2.4 Raisonner, c'est prouver

Les ensembles engendrés inductivement et co-inductivement par un système d'inférence ont été caractérisés en tant que points fixes de l'opérateur d'inférence associé et en tant que conclusions de preuves. On va maintenant associer à ces caractérisations des principes pour raisonner sur les éléments de ces ensembles engendrés. Fidèle à notre ligne directrice, nous insistons sur la formulation en termes de preuves. Ainsi, un raisonnement co-inductif correspond à la validation d'un jugement par une preuve : raisonner, c'est donc prouver ; un raisonnement inductif s'appuie sur la propriété des preuves admissibles considérées, la bonne fondation : il s'agit donc d'une récurrence structurelle sur les preuves.
On considère tout d'abord le cas d'un système d'inférence, puis celui d'un treillis complet. À chaque fois, on donne tout d'abord les principes d'induction, puis ceux de co-induction. On compare aussi les différents principes, relativement à ceux utilisant les preuves. Pour simplifier, on utilise les preuves canoniquement associées aux systèmes d'inférence considérés.
Les principes donnés sont toujours optimaux, dans le sens où la condition permettant de déduire la conclusion souhaitée lui est en fait équivalente.

Considérons tout d'abord le cas d'un système d'inférence. Les différentes caractérisations données dans le paragraphe 1.2.1 mènent au théorème suivant pour l'induction.

1.2.10 Théorème (Système d'inférence : principes inductifs)
Considérons un système d'inférence Φ *défini sur l'ensemble* $\mathcal{U}$*. Soient* φ *l'opérateur d'inférence associé,* $(\Delta_\alpha(\varphi))_\alpha$ *la suite transfinie croissante des itérés de* φ*, définie par*

$$\Delta_\alpha(\varphi) = \bigcup_{\beta|\beta<\alpha} \varphi(\Delta_\beta(\varphi))\,,$$

$\mathcal{P}_\Delta(\Phi)$ *l'ensemble des preuves dans* Φ *bien fondées et* $\Delta(\Phi)$ *l'ensemble engendré inductivement par* Φ*.*
Soit Z *une partie de* $\mathcal{U}$*. Alors chacune des conditions suivantes implique que* $\Delta(\Phi)$ *est inclus dans* Z*, et réciproquement :*

- (i) *conditions de stabilité :*
  (a) $\varphi(Z \cap \Delta(\Phi)) \subseteq Z$,
  (b) $\exists\, Y\,.\ Y \subseteq Z \wedge \varphi(Y) \subseteq Y$,
- (ii) *condition d'induction transfinie :*
  $\forall\, \alpha\,.\ (\forall\, \beta < \alpha\,.\ \Delta_\beta(\varphi) \subseteq Z) \Rightarrow \Delta_\alpha(\varphi) \subseteq Z$,
- (iii) *condition de récurrence structurelle sur les preuves :*
  $\forall\, a \in \mathcal{P}_\Delta(\Phi)\,.\ (\forall\, j \in \operatorname{dom} a \cap \mathcal{U}\,.\ a(j) \in Z) \Rightarrow a(\varepsilon) \in Z\,.$

Démonstration
Notons (C) la conclusion $\Delta(\Phi) \subseteq Z$.
• $(i.a) \Rightarrow (C)$
Supposons $(i.a)$. Comme $\Delta(\Phi)$ est un point fixe, $Z \cap \Delta(\Phi)$ est stable par φ. Du théorème de Tarski (1.2.1 (p. 69)), on déduit $\Delta(\Phi) \subseteq Z \cap \Delta(\Phi)$, d'où (C).
• $(i.b) \Rightarrow (C)$
Supposons $(i.b)$. Comme Y est stable, par le théorème de Tarski (1.2.1 (p. 69)), $\Delta(\Phi) \subseteq Y$, puis par transitivité (C).
• $(ii) \Rightarrow (C)$
Supposons (ii). On déduit du principe d'induction transfinie que pour tout ordinal α, $\Delta_\alpha(\varphi) \subseteq Z$, d'où (C) par le théorème 1.2.2 (p. 70).
• $(iii) \Rightarrow (C)$
Supposons (iii). Du principe de récurrence structurelle sur les preuves bien fondées, on déduit $\forall\, a \in \mathcal{P}_\Delta(\Phi)\,.\; a(\varepsilon) \in Z$. Par le théorème 1.2.5 (p. 77), on a alors (C).
• Réciproques
Elles sont triviales.
⌞

D'un point de vue logique, toutes ces conditions sont équivalentes, comme elles sont équivalentes à la conclusion du principe, l'inclusion de l'ensemble engendré inductivement dans la partie considérée. Il est cependant intéressant de les comparer d'un point de vue méthodologique. À cette fin, on se place dans une position quelque peu artificielle : on cherche à démontrer la condition de récurrence structurelle sur les preuves, en supposant tour à tour chacune des autres conditions, comme si nous ignorions, malgré l'évidence, que le principe d'induction peut s'appliquer à chaque fois.
La première condition de stabilité exprime de manière synthétique la condition de récurrence structurelle sur les preuves : il s'agit de sa traduction utilisant l'opérateur d'inférence. Quant à la seconde condition de stabilité, elle conduit à démontrer la condition de récurrence avec Y, et non Z, puis à utiliser l'inclusion de Y dans Z.
La condition d'induction transfinie ressemble à la condition de récurrence structurelle. Pour préciser leur rapport, nous allons caractériser les itérés de l'opérateur d'inférence à l'aide de preuves. Considérons les premiers termes de la suite des itérés, $(\Delta_\alpha(\varphi))_\alpha$, avec la notation du théorème précédent :
- $\Delta_0(\varphi)$ est l'ensemble vide,
- $\Delta_1(\varphi)$ est l'ensemble des axiomes,
- $\Delta_2(\varphi)$ est l'ensemble formé des axiomes et des conclusions de règles

dont les prémisses sont des axiomes.

Ces premières caractérisations suggèrent que l'ordinal indique la hauteur des preuves, intuition que nous développons maintenant.

La hauteur d'un arbre, entendue comme la plus grande longueur d'un chemin entre la racine et une feuille de l'arbre, est définie lorsque l'arbre est fini : on peut cependant étendre la définition aux arbres bien fondés, à condition de renoncer à l'ensemble des entiers naturels pour la classe des ordinaux.

1.2.11 Définition et proposition (**Hauteur (ordinale) d'un arbre bien fondé**)

Il existe une unique application h *de l'ensemble des arbres bien fondés dans la classe des ordinaux vérifiant pour tout arbre $f(a_i)_{i\in I}$ d'étiquette f et de sous-arbres les arbres de la famille $(a_i)_{i\in I}$:*

$$\mathrm{h}(f(a_i)_{i\in I}) = \bigvee_{i\in I} \mathrm{h}(a_i) + 1\,.$$

L'ordinal $\mathrm{h}(f(a_i)_{i\in I})$ *est la* hauteur *de l'arbre $f(a_i)_{i\in I}$.*

L'existence et l'unicité découle du théorème 1.1.1 (p. 47) concernant la récursion sur les ensembles munis d'une relation bien fondée[26].

Appliquons cette notion de hauteur aux preuves. Considérons un système d'inférence Φ et un jugement z appartenant à l'ensemble engendré inductivement par Φ. On associe à z le plus petit ordinal de l'ensemble des hauteurs des preuves (bien fondées) validant z. Cet ordinal, non nul, est appelé la *complexité* du jugement z dans Φ.

Finalement, les conditions d'induction transfinie et de récurrence structurelle sur les preuves se correspondent très précisément.

1.2.12 Proposition (Induction : itérés et complexité)

Considérons un système d'inférence Φ sur l'ensemble $\mathcal{U}$.

Soient φ l'opérateur d'inférence associé et $(\Delta_\alpha(\varphi))_\alpha$ la suite transfinie croissante des itérés de φ, définie par

$$\Delta_\alpha(\varphi) = \bigcup_{\beta|\beta<\alpha} \varphi(\Delta_\beta(\varphi))\,.$$

Alors, pour tout jugement z et pour tout ordinal α,

[26] Le théorème a été établi dans le cas où l'ensemble d'arrivée est un ensemble (de la théorie des ensembles), et non une classe propre (ce qu'est la classe des ordinaux). On admet qu'il est encore valable dans ce cas.

z appartient à $\Delta_\alpha(\varphi)$ si et seulement si la complexité de z dans Φ est inférieure ou égale à α.

Démonstration
Soit R_α l'ensemble des jugements de complexité inférieure ou égale à α.
• $\Delta_\alpha(\varphi) \subseteq R_\alpha$
On procède par induction transfinie.
Soit α un ordinal tel que $\forall\, \beta < \alpha\,.\ \Delta_\beta(\varphi) \subseteq R_\beta$, et considérons $z \in \Delta_\alpha(\varphi)$. Il existe $\beta < \alpha$ tel que $z \in \varphi(\Delta_\beta(\varphi))$. Cela signifie qu'il existe une règle (Z, z) de Φ, avec $Z \subseteq \Delta_\beta(\varphi)$, soit par hypothèse d'induction, $Z \subseteq R_\beta$. Tout jugement j de Z a ainsi une complexité inférieure ou égale à β, si bien que la complexité de z est inférieure ou égale à $\beta + 1$, et donc à α.
• $R_\alpha \subseteq \Delta_\alpha(\varphi)$
On procède par induction transfinie.
Soit α un ordinal tel que $\forall\, \beta < \alpha\,.\ R_\beta \subseteq \Delta_\beta(\varphi)$, et considérons $z \in R_\alpha$, de complexité β, inférieure ou égale à α.
Si $\beta < \alpha$, alors l'hypothèse d'induction donne $z \in \Delta_\beta(\varphi)$ et comme $\Delta_\beta(\varphi) \subseteq \Delta_\alpha(\varphi)$, on peut conclure.
Supposons donc $\beta = \alpha$. Comme β est la hauteur d'un arbre, il existe β' tel que $\beta = \beta' + 1$. Soit (Z, z) la règle d'inférence par laquelle se termine la preuve de z de hauteur β. Nécessairement, les jugements de Z ont une complexité inférieure ou égale à β'. De l'hypothèse d'induction, on déduit que $Z \subseteq \Delta_{\beta'}(\varphi)$, ce qui montre que z appartient à $\varphi(\Delta_{\beta'}(\varphi))$, et donc à $\Delta_\alpha(\varphi)$.
⌞

Toujours au sein d'un système d'inférence, voyons maintenant la co-induction, à partir des caractérisations données dans le paragraphe 1.2.1.

1.2.13 Théorème (Système d'inférence : principes co-inductifs)
Considérons un système d'inférence Φ sur l'ensemble $\mathcal{U}$.
Soient φ l'opérateur d'inférence associé, $(\nabla_\alpha(\varphi))_\alpha$ la suite transfinie décroissante des itérés de φ, définie par

$$\nabla_\alpha(\varphi) = \bigcap_{\beta|\beta<\alpha} \varphi(\nabla_\beta(\varphi))\,,$$

$\nabla(\Phi)$ l'ensemble engendré co-inductivement par Φ et $\mathcal{P}_\nabla(\Phi)$ l'ensemble des preuves dans Φ.
Soit Z une partie de $\mathcal{U}$. Alors chacune des conditions suivantes implique que

Z est inclus dans $\nabla(\Phi)$, et réciproquement :

(i) *conditions de densité :*
$(a)\ Z \subseteq \varphi(Z \cup \nabla(\Phi))\,,$
$(b)\ \exists Y\,.\ Z \subseteq Y \wedge Y \subseteq \varphi(Y)\,,$
(ii) *condition d'induction transfinie :*
$\forall \alpha\,.\ (\forall \beta < \alpha\,.\ Z \subseteq \nabla_\beta(\varphi)) \Rightarrow Z \subseteq \nabla_\alpha(\varphi)\,,$
(iii) *validation par preuve :*
$\forall z \in Z\,.\ \exists a \in \mathcal{P}_\nabla(\Phi)\,.\ a(\varepsilon) = z\,.$

Démonstration
Notons (C) la conclusion $Z \subseteq \nabla(\Phi)$.
• $(i.a) \Rightarrow (C), (i.b) \Rightarrow (C), (ii) \Rightarrow (C)$
En raisonnant par dualité, on peut utiliser les résultats du théorème 1.2.10 (p. 88).
• $(iii) \Rightarrow (C)$
Supposons (iii). Du théorème 1.2.5 (p. 77), on déduit immédiatement (C).
• Réciproques
Elles sont triviales.
⌞

Comme pour l'induction, toutes ces conditions sont équivalentes, comme elles sont équivalentes à la conclusion du principe, l'inclusion de Z dans l'ensemble engendré co-inductivement par le système d'inférence. Il reste la comparaison méthodologique : cette fois-ci, on cherche à valider par des preuves les jugements de Z. Examinons une à une les autres conditions et déterminons la manière dont chacune d'elles permet cette validation.
Chaque condition de densité permet de construire un système d'équations récursives dont la solution donne les preuves recherchées. En effet, dans les deux cas, il est possible de définir un système d'équations récursives, quasi-uniforme et compatible avec Φ, $(x_j = a_j)_{j \in J}$, vérifiant
– $J \subseteq \mathcal{U}$,
– $Z \subseteq J$,
– et $a_j(\varepsilon) = j$.
La première condition, $Z \subseteq \varphi(Z \cup \nabla(\Phi))$, permet de définir un système avec $J = Z \cup \nabla(\Phi)$; pour la seconde, $\exists Y\,.\ Z \subseteq Y \wedge Y \subseteq \varphi(Y)$, on prend évidemment J vérifiant $Z \subseteq J \wedge J \subseteq \varphi(J)$. Dans tous les cas, pour tout jugement j de J, donc de Z, la valeur de la solution en x_j est une preuve de j.
Venons-en à la condition d'induction transfinie. La suite des itérés $(\nabla_\alpha(\varphi))_\alpha$ est décroissante et converge vers le plus grand point fixe. Étant donné un

itéré $\nabla_\alpha(\varphi)$ de l'opérateur d'inférence φ, essayons de valider par des preuves ses jugements. Nécessairement, le système d'inférence considéré doit être une extension de Φ ; le plus simple est d'ajouter des axiomes. Avant d'étudier les premiers termes de la suite pour en induire une caractérisation générale, remarquons qu'en raisonnant par dualité, il est facile de valider par des preuves les jugements du complémentaire de $\nabla_\alpha(\varphi)$.
L'*opérateur dual* de φ, noté $\widetilde{\varphi}$, est défini par

$$\widetilde{\varphi}(Z) = \neg\varphi(\neg Z)\,,$$

où pour toute partie Z de l'ensemble des jugements, $\neg Z$ est son complémentaire. Le plus petit point fixe de φ a pour complémentaire le plus grand point fixe de $\widetilde{\varphi}$, et symétriquement pour le plus grand. De même, les suites transfinies des itérés de φ et $\widetilde{\varphi}$ se correspondent : pour tout ordinal α, les itérés $\Delta_\alpha(\varphi)$ et $\nabla_\alpha(\varphi)$ ont pour complémentaire $\nabla_\alpha(\widetilde{\varphi})$ et $\Delta_\alpha(\widetilde{\varphi})$ respectivement.
Le *système d'inférence dual* de Φ, noté $\widetilde{\Phi}$, se définit à partir de $\widetilde{\varphi}$, et contient donc toute règle de la forme (Z, z) telle que pour toute règle de Φ ayant pour conclusion z, au moins une de ses prémisses appartient à Z :

$$\forall\,(Y, z) \in \Phi\,.\ Z \cap Y \neq \emptyset\,.$$

C'est qu'en effet on a les équivalences suivantes :

$$\begin{aligned}
(Z, z) \in \widetilde{\Phi} \quad &\Leftrightarrow \quad z \in \widetilde{\varphi}(Z) \\
&\Leftrightarrow \quad z \notin \varphi(\neg Z) \\
&\Leftrightarrow \quad \neg(\exists\, Y\,.\ (Y, z) \in \Phi \wedge Y \cap Z = \emptyset) \\
&\Leftrightarrow \quad \forall\, Y\,.\ (Y, z) \in \Phi \Rightarrow Z \cap Y \neq \emptyset\,.
\end{aligned}$$

Il s'ensuit, d'après la proposition 1.2.12 (p. 90), que le complémentaire de $\nabla_\alpha(\varphi)$ est égal à l'ensemble des jugements de complexité inférieure ou égale à α dans $\widetilde{\Phi}$.
En dehors du plus grand point fixe de φ, l'itéré $\nabla_\alpha(\varphi)$ contient des jugements appartenant au plus petit point fixe de $\widetilde{\varphi}$: leur complexité est alors supérieure à α dans $\widetilde{\Phi}$. Affinons ce résultat en considérant les deux premiers termes de la suite.
On a $\nabla_0(\varphi) = \mathcal{U}$. Considérons un jugement z, et essayons de le prouver dans une extension de Φ. S'il est conclusion d'une règle de Φ, on examine chacune des prémisses de la règle ; si cette prémisse est conclusion d'une règle, on peut continuer de même, sinon, il est nécessaire d'ajouter comme axiome ce jugement. C'est même suffisant puisqu'ainsi aussi bien les conclusions des

règles que les jugements qui n'en sont pas peuvent être prouvés. Comme les axiomes de $\widetilde{\Phi}$ sont les jugements qui ne sont pas conclusion d'une règle de Φ, il s'agit donc de les ajouter, autrement dit d'ajouter les jugements de complexité un dans $\widetilde{\Phi}$.
On a $\nabla_1(\varphi) = \varphi(\mathcal{U})$. C'est donc l'ensemble des conclusions des règles de Φ. Essayons de prouver une conclusion z d'une règle de Φ. Soit $((Z_i, z))_{i\in I}$ la famille formée des règles de Φ ayant pour conclusion z. S'il existe $i \in I$ tel que Z_i ne contient que des conclusions de règles, on peut continuer en prenant comme prémisses de z l'ensemble Z_i. Sinon, pour chaque i, il existe z_i dans Z_i qui n'est pas conclusion d'une règle. Il est facile de voir que la règle $(\{z_i \mid i \in I\}, z)$ appartient alors au système dual $\widetilde{\Phi}$. Le jugement z, qui n'est pas un axiome de $\widetilde{\Phi}$, a donc pour complexité deux dans $\widetilde{\Phi}$.
Généralisons avec la proposition suivante.

1.2.14 Proposition (Co-induction : itérés et complexité)
Soient Φ un système d'inférence sur l'ensemble $\mathcal{U}$, φ l'opérateur associé, $(\nabla_\alpha(\varphi))_\alpha$ la suite transfinie décroissante des itérés de φ, définie par

$$\nabla_\alpha(\varphi) = \bigcap_{\beta|\beta<\alpha} \varphi(\nabla_\beta(\varphi))\,,$$

et enfin $\widetilde{\Phi}$ le système d'inférence dual de Φ.
Soit α un ordinal. Définissons le système d'inférence Φ_α comme une extension de Φ, contenant en plus les axiomes formés des jugements de complexité $\alpha+1$ dans $\widetilde{\Phi}$:

$$\Phi_\alpha = \Phi \cup \{(\emptyset, z) \mid z \text{ de complexité } \alpha+1 \text{ dans } \widetilde{\Phi}\}\,.$$

Alors $\nabla_\alpha(\varphi)$ est égal à l'ensemble engendré co-inductivement par Φ_α.

Démonstration
• $\nabla_\alpha(\varphi) \subseteq \nabla(\Phi_\alpha)$
On va construire un système d'équations récursives, d'inconnues $(x_z)_{z\in\nabla_\alpha(\varphi)}$, dont la solution définit pour chaque z de $\nabla_\alpha(\varphi)$ une preuve dans Φ_α.
Soit $z \in \nabla_\alpha(\varphi)$.
Pour définir l'équation en x_z, considérons la famille $((Z_i, z))_{i\in I}$ formée des règles de Φ ayant pour conclusion z, et distinguons deux cas.
◦ $\exists\, i \in I\,.\ \forall\, j \in Z_i\,.\ j \in \nabla_\alpha(\varphi)$
Soit i appartenant à I vérifiant $\forall\, j \in Z_i\,.\ j \in \nabla_\alpha(\varphi)$. On pose :

$$x_z = \frac{(x_j)_{j\in Z_i}}{z}\,.$$

◦ $\forall\, i \in I\,.\ \exists\, j \in Z_i\,.\ j \notin \nabla_\alpha(\varphi)$
On peut donc associer à tout $i \in I$ un jugement z_i de Z_i vérifiant $z_i \notin \nabla_\alpha(\varphi)$, soit $z_i \in \Delta_\alpha(\widetilde{\varphi})$. Il est facile de vérifier que la règle $(\{z_i \mid i \in I\}, z)$ est une règle du système dual $\widetilde{\Phi}$. Comme les z_i ont une complexité dans $\widetilde{\Phi}$ d'au plus α, z y a donc une complexité d'au plus $\alpha + 1$. Comme $z \notin \Delta_\alpha(\widetilde{\varphi})$ par hypothèse, il en résulte que z a pour complexité $\alpha + 1$ dans $\widetilde{\Phi}$. Finalement, on pose :

$$x_z = \frac{\emptyset}{z}.$$

Le système ainsi construit est quasi-uniforme et compatible avec Φ_α ; d'après la proposition 1.2.3 (p. 73), pour tout z appartenant à $\nabla_\alpha(\varphi)$, la valeur de la solution en x_z est donc une preuve de z dans Φ_α.
• $\nabla(\Phi_\alpha) \subseteq \nabla_\alpha(\varphi)$
On procède par induction transfinie.
Soit α un ordinal et supposons $\forall\, \beta < \alpha\,.\ \nabla(\Phi_\beta) \subseteq \nabla_\beta(\varphi)$. Du fait de l'inclusion précédente, on a en fait $\forall\, \beta < \alpha\,.\ \nabla(\Phi_\beta) = \nabla_\beta(\varphi)$.
On montre que pour tout ordinal $\beta < \alpha$, on a $\nabla(\Phi_\alpha) \subseteq \varphi(\nabla_\beta(\varphi))$, ce qui permettra de conclure.
Soit donc $\beta < \alpha$. On va construire un système d'équations récursives, d'inconnues $((x_z)_{z\in\nabla(\Phi_\alpha)}, (y_j)_{j\in\nabla(\Phi_\beta)})$, dont la solution définit pour chaque z de $\nabla(\Phi_\alpha)$ une preuve de z dans Φ_β se terminant par l'application d'une règle de Φ. On en déduira que $\nabla(\Phi_\alpha) \subseteq \varphi(\nabla(\Phi_\beta))$, ce qui est équivalent par l'hypothèse d'induction, à $\nabla(\Phi_\alpha) \subseteq \varphi(\nabla_\beta(\varphi))$.
Soit j appartenant à $\nabla(\Phi_\beta)$. Il admet dans Φ_β une preuve a_j, et on pose :

$$y_j = a_j\,.$$

Soit z appartenant à $\nabla(\Phi_\alpha)$. Distinguons deux cas.
◦ z a pour complexité $\alpha + 1$ dans le système dual $\widetilde{\Phi}$.
Considérons la famille $((Z_i, z))_{i\in I}$ formée des règles de Φ ayant pour conclusion z, et montrons par l'absurde qu'il existe $i \in I$ tel que $Z_i \subseteq \nabla_\beta(\varphi)$.
Supposons $\forall\, i \in I\,.\ \exists\, z \in Z_i\,.\ z \notin \nabla_\beta(\varphi)$. Comme précédemment, on montre que z a alors une complexité dans $\widetilde{\Phi}$ d'au plus $\beta + 1$, donc d'au plus α, contradiction.
Soit $i \in I$ tel que $Z_i \subseteq \nabla_\beta(\varphi)$, soit par hypothèse d'induction $Z_i \subseteq \nabla(\Phi_\beta)$. L'équation en x_z est définie par :

$$x_z = \frac{(y_j)_{j\in Z_i}}{z}.$$

◦ z n'a pas pour complexité $\alpha + 1$ dans $\widetilde{\Phi}$.
Une preuve de z dans Φ_α se termine donc par l'application d'une règle de

Φ, soit (Z, z), avec $Z \subseteq \nabla(\Phi_\alpha)$, et on pose :

$$x_z = \frac{(x_j)_{j \in Z}}{z}.$$

Le système ainsi construit est quasi-uniforme et compatible avec Φ_β. D'après la proposition 1.2.3 (p. 73), pour tout z appartenant à $\nabla(\Phi_\alpha)$, la valeur de la solution en x_z est une preuve de z dans Φ_β, se terminant par l'application d'une règle de Φ ; comme les prémisses de z appartiennent à $\nabla(\Phi_\beta)$, on en déduit que z appartient à $\varphi(\nabla(\Phi_\beta))$, soit par l'hypothèse d'induction à $\varphi(\nabla_\beta(\varphi))$.
⌞

Revenons finalement au problème initial, la validation des jugements d'une partie Z vérifiant la condition d'induction transfinie, et donc pour tout ordinal α, l'inclusion dans $\nabla_\alpha(\varphi)$.
Soit α un ordinal.
En utilisant la même technique que dans la preuve précédente, on construit un système d'équations récursives, d'inconnues $(x_z)_{z \in \nabla_\alpha(\varphi)}$. Soit z appartenant à $\nabla_\alpha(\varphi)$, et considérons la famille $((Z_i, z))_{i \in I}$ formée des règles de Φ de conclusion z. S'il existe i tel que $Z_i \subseteq \nabla_\alpha(\varphi)$, alors on pose :

$$x_z = \frac{(x_j)_{j \in Z_i}}{z}.$$

Sinon, pour chaque i, il existe z_i dans Z_i n'appartenant pas à $\nabla_\alpha(\varphi)$. Comme précédemment, il est facile de voir que z a pour complexité $\alpha + 1$ dans $\widetilde{\Phi}$, et on pose :

$$x_z = \frac{\emptyset}{z}.$$

Le système étant uniforme et compatible avec Φ_α, chaque jugement de $\nabla_\alpha(\varphi)$, et donc de Z, admet une preuve dans Φ_α. Pour α suffisamment grand, $\Phi = \Phi_\alpha$, si bien qu'on obtient une preuve dans Φ.

Donnons pour terminer les principes d'induction et de co-induction dans un treillis complet. On s'appuie sur les caractérisations obtenues dans le paragraphe 1.2.3 (p. 83).
On peut énoncer pour l'induction le théorème suivant, très proche du théorème 1.2.10 (p. 88) pour les systèmes d'inférence.

1.2.15 Théorème (Treillis complet : principes inductifs)
Soit $(\mathcal{U}, \leq, \vee, \wedge, \bot, \top)$ un treillis complet. Soient η une application croissante de $\mathcal{U}$ dans lui-même, et $(\Delta_\alpha(\eta))_\alpha$ la suite transfinie croissante des itérés de

η, définie par

$$\Delta_\alpha(\eta) = \bigvee_{\beta|\beta<\alpha} \eta(\Delta_\beta(\eta))\,.$$

Associons à η le système d'inférence Φ sur $\mathcal{U}$, formé des règles de la forme :

$$\frac{Z}{z} \quad (Z \subseteq \mathcal{U}, z \leq \eta(\vee Z))\,,$$

et le système Φ', formé des règles de la forme :

$$\frac{Z}{\eta(\vee Z)} \quad (Z \subseteq \mathcal{U})\,.$$

Soient enfin $\mathrm{lfp}(\eta)$ *le plus petit point fixe de η, et $\mathcal{P}_\Delta(\Phi)$ et $\mathcal{P}_\Delta(\Phi')$ les ensembles des preuves bien fondées dans Φ et Φ' respectivement.*
Considérons un jugement z. Alors les conditions suivantes impliquent que $\mathrm{lfp}(\eta)$ *est inférieur ou égal à z, et réciproquement :*

- (i) *conditions de stabilité :*
  - (a) $\eta(z \wedge \mathrm{lfp}(\eta)) \leq z$,
  - (b) $\exists\, y\,.\ y \leq z \wedge \eta(y) \leq y$,
- (ii) *condition d'induction transfinie :*
  $\forall\, \alpha\,.\ (\forall\, \beta < \alpha\,.\ \Delta_\beta(\eta) \leq z) \Rightarrow \Delta_\alpha(\eta) \leq z$,
- (iii) *conditions de récurrence structurelle sur les preuves :*
  - (a) $\forall\, a \in \mathcal{P}_\Delta(\Phi)\,.\ (\forall\, j \in \mathrm{dom}\, a \cap \mathcal{U}\,.\ a(j) \leq z) \Rightarrow a(\varepsilon) \leq z$,
  - (b) $\forall\, a \in \mathcal{P}_\Delta(\Phi')\,.\ (\forall\, j \in \mathrm{dom}\, a \cap \mathcal{U}\,.\ a(j) \leq z) \Rightarrow a(\varepsilon) \leq z$.

Démonstration
Notons (C) la conclusion $\mathrm{lfp}(\eta) \leq z$.
• $(i.a) \Rightarrow (C)$
Supposons $(i.a)$. Comme $\mathrm{lfp}(\eta)$ est un point fixe, $z \wedge \mathrm{lfp}(\eta)$ est stable par η. Du théorème de Tarski (1.2.1 (p. 69)), on déduit $\mathrm{lfp}(\eta) \leq z \wedge \mathrm{lfp}(\eta)$, d'où (C).
• $(i.b) \Rightarrow (C)$
Supposons $(i.b)$. Comme y est stable, par le théorème de Tarski (1.2.1 (p. 69)), $\mathrm{lfp}(\eta) \leq y$, puis par transitivité (C).
• $(ii) \Rightarrow (C)$
Supposons (ii). On déduit du principe d'induction transfinie que pour tout ordinal α, $\Delta_\alpha(\eta) \leq z$, d'où (C) par le théorème 1.2.2 (p. 70).
• $(iii.a) \Rightarrow (C)$
Supposons (iii). Du principe de récurrence structurelle sur les preuves bien

fondées, on déduit $\forall\, a \in \mathcal{P}_\Delta(\Phi)\,.\ a(\varepsilon) \leq z$. Par le théorème 1.2.8 (p. 85), on a alors (C).
• $(iii.b) \Rightarrow (C)$
C'est le même raisonnement, cette fois en utilisant le théorème 1.2.9 (p. 86).
• Réciproques
Elles sont triviales.
⌞

Pour la co-induction, on dispose des principes suivants.

1.2.16 Théorème (Treillis complet : principes co-inductifs)
Soit $(\mathcal{U}, \leq, \vee, \wedge, \bot, \top)$ un treillis complet. Soient η une application croissante de $\mathcal{U}$ dans lui-même, et $(\nabla_\alpha(\eta))_\alpha$ la suite transfinie décroissante des itérés de η, définie par

$$\nabla_\alpha(\eta) = \bigwedge_{\beta|\beta<\alpha} \eta(\nabla_\beta(\eta))\,.$$

Associons à η le système d'inférence Φ sur $\mathcal{U}$, formé des règles de la forme :

$$\frac{Z}{z} \quad (Z \subseteq \mathcal{U}, z \leq \eta(\vee Z))\,,$$

et le système Φ', formé des règles de la forme :

$$\frac{Z}{\eta(\vee Z)} \quad (Z \subseteq \mathcal{U})\,.$$

Soient enfin $\mathrm{gfp}(\eta)$ le plus grand point fixe de η, et $\mathcal{P}_\nabla(\Phi)$ et $\mathcal{P}_\nabla(\Phi')$ les ensembles des preuves dans Φ et Φ' respectivement.
Considérons un jugement z. Alors les conditions suivantes impliquent que $\mathrm{gfp}(\eta)$ est supérieur ou égal à z, et réciproquement :

- (i) *conditions de densité :*
  - (a) $z \leq \eta(z \vee \mathrm{gfp}(\eta))\,,$
  - (b) $\exists\, y\,.\ z \leq y \wedge y \leq \eta(y)\,,$
- (ii) *condition d'induction transfinie :*
  $\forall\, \alpha\,.\ (\forall\, \beta < \alpha\,.\ z \leq \nabla_\beta(\eta)) \Rightarrow z \leq \nabla_\alpha(\eta)\,,$
- (iii) *validation par preuve :*
  - (a) $\exists\, a \in \mathcal{P}_\nabla(\Phi)\,.\ a(\varepsilon) = z\,,$
  - (b) $\exists\, y \in \mathcal{U}\,.\ \exists\, a \in \mathcal{P}_\nabla(\Phi')\,.\ z \leq y \wedge a(\varepsilon) = y\,.$

Démonstration
Notons (C) la conclusion $z \leq \mathrm{gfp}(\eta)$.

• $(i.a) \Rightarrow (C)$
Supposons $(i.a)$. Comme $\mathrm{gfp}(\eta)$ est un point fixe, $z \vee \mathrm{gfp}(\eta)$ est dense pour η. Du théorème de Tarski (1.2.1 (p. 69)), on déduit $z \vee \mathrm{gfp}(\eta) \leq \mathrm{gfp}(\eta)$, d'où (C).
• $(i.b) \Rightarrow (C)$
Supposons $(i.b)$. Comme y est dense, par le théorème de Tarski (1.2.1 (p. 69)), $y \leq \mathrm{gfp}(\eta)$, puis par transitivité (C).
• $(ii) \Rightarrow (C)$
Supposons (ii). On déduit du principe d'induction transfinie que pour tout ordinal α, $z \leq \nabla_\alpha(\eta)$, d'où (C) par le théorème 1.2.2 (p. 70).
• $(iii.a) \Rightarrow (C)$
Supposons $(iii.a)$. Du théorème 1.2.8 (p. 85), on déduit (C).
• $(iii.b) \Rightarrow (C)$
Cette fois, on utilise le théorème 1.2.9 (p. 86).
• Réciproques
Elles sont triviales.
⌞

Comme précédemment, il serait possible de comparer les principes d'un point de vue méthodologique. C'est cependant inutile dans la mesure où il est facile de traduire les résultats précédents dans ce nouveau contexte. En effet, en associant à l'application η les opérateurs d'inférence des systèmes Φ et Φ', notés φ et φ', on peut traduire toutes les conditions portant sur η en des conditions analogues utilisant φ ou φ'. Par exemple, les itérés de η, de φ et de φ' vérifient pour tout ordinal α :

$$\Delta_\alpha(\eta) = \vee\Delta_\alpha(\varphi) = \vee\Delta_\alpha(\varphi') \,.$$

L'interprétation en termes de preuves dans Φ ou Φ' en découle.

Nous avons essayé de montrer dans ce chapitre que l'approche déductive est une alternative viable à l'approche classique par points fixes. Elle repose sur l'utilisation de systèmes d'inférence et s'appuie sur une pratique unique, la construction de preuves dans les systèmes d'inférence.
Dans les chapitres suivants, nous construirons des objets infinis en utilisant des systèmes d'équations récursives et raisonnerons sur ces objets en suivant cette approche déductive.

Chapitre 2

La sémantique observationnelle

Sommaire

Dans ce chapitre, nous répondons à la question suivante : étant donné un programme, que peut-on en observer ?
C'est un préalable à notre étude des flux d'informations, présentée au chapitre suivant. La généralité de la question nous a conduit à développer ce chapitre sans référence à notre problématique sécuritaire, mais plutôt sous l'angle purement sémantique.

Ainsi, on peut considérer que le but de ce chapitre est de développer une méthode pour définir la sémantique de langages de programmation algorithmique. Elle est présentée ici pour le langage paradigmatique par excellence,

le λ-calcul paresseux[1] ou faible, dans sa version utilisant l'appel par valeur. La simplicité de ce langage rend l'exposé plus clair, mais peut paraître restrictive ; que devient la méthode présentée ici lorsqu'on étend le langage, syntaxiquement en structurant les données, mais aussi sémantiquement, en introduisant des effets de bord, des entrées et sorties, etc. ? C'est en conclusion qu'on abordera cette question de la généralisation.

Les différentes approches sémantiques peuvent se partager suivant la manière dont l'exécution est décrite :

- ou bien elle l'est effectivement, par la suite des opérations qui mènent au résultat, et la sémantique est dite opérationnelle,
- ou bien elle ne l'est pas, seul le résultat important, indépendamment de la manière dont le calcul s'est réalisé, et on parle de sémantique dénotationnelle.

Une *sémantique opérationnelle* décrit ainsi l'exécution des programmes sur une machine, généralement abstraite : elle associe à chaque programme sa trace d'exécution, soit la suite des états successifs de la machine abstraite lors de l'exécution. De ce point de vue, elle est proche de l'activité de programmation à laquelle elle donne un sens.

Une sémantique opérationnelle peut servir à définir ce qu'on observe d'un programme, par expérimentation : il suffit d'exécuter le programme dans tout contexte d'utilisation possible et d'observer le résultat qu'il donne. Il est à noter que cette définition peut être circulaire, puisque l'observation d'un programme se définit à partir de l'observation de résultats d'exécution. Deux remèdes à la circularité : d'une part, l'accepter, ce qui revient à définir l'observation co-inductivement, d'autre part, définir l'observation d'un résultat de manière élémentaire, par sa valeur si le résultat appartient à un type de base, dit « passif », comme un entier, un booléen, ou par l'observation de sa terminaison s'il appartient à un type « actif », représentant des fonctions ou des objets.

Étant donné un programme, quelle information apporte sa trace d'exécution sur son observation ? Il est clair que les états intermédiaires ne sont pas observables. Si la trace est infinie, on peut considérer qu'il est inutile d'expérimenter avec ce programme divergent : l'observation de la divergence suffit. Si la trace est finie, alors le programme donne un résultat final ; si ce résultat appartient à un type « passif », on peut se limiter à l'observation de sa valeur : l'expérimentation n'apporte rien à ce qu'on peut observer du

[1] « Paresseux » signifie ici que l'évaluation s'arrête sur les abstractions, leurs corps n'étant donc pas évalués, contrairement au λ-calcul dans sa théorie classique. Une acception courante, que nous ne suivons pas, y associe aussi l'appel par nom.

programme ; si ce résultat appartient à un type « actif », sa connaissance ne donne pas directement d'informations sur son observation : il est nécessaire d'expérimenter. Dans la mesure où nous nous intéressons aux langages d'ordre supérieur, comme le λ-calcul, seuls capables de modéliser des langages évolués, la sémantique opérationnelle atteint ici sa limite.
Une *sémantique dénotationnelle* doit permettre de résoudre cette limitation, en associant à chaque programme une dénotation, typiquement une fonction décrite en extension, sans référence à la manière dont le calcul peut se réaliser[2]. Le domaine dénotationnel doit caractériser alors ce qui est calculable par un programme, et cette caractérisation, difficile, fait moins appel à l'intuition du programmeur qu'à celle du mathématicien. Dans les faits, cette rupture se traduit par l'absence de sémantiques dénotationnelles dans les spécifications de langages très répandus, comme le langage orienté objets Java : seul l'aspect opérationnel est traité, supposé mieux adapté aux lecteurs programmeurs. C'est un premier inconvénient de la sémantique dénotationnelle, pratique. S'y ajoutent deux autres, plus théoriques : d'une part, si l'objectif de caractériser la calculabilité a pu être atteint dans des cas élémentaires, il est difficile de généraliser la solution à des exemples plus complets, comme nous l'évoquerons en conclusion, d'autre part, et c'est ce qui nous intéresse particulièrement, cet objectif n'est pas toujours atteint au mieux, ne résolvant qu'incomplètement la limitation des sémantiques opérationnelles. C'est un échec par le haut, par une générosité excessive, pour reprendre le mot de Milner, dans l'introduction de [53] : les domaines dénotationnels sont trop riches et permettent de distinguer des programmes qui pourtant observationnellement sont semblables.
Notre but est double :

- réconcilier le programmeur avec la sémantique dénotationnelle, en la construisant simplement à partir de la sémantique opérationnelle,
- parvenir à résoudre complètement, grâce à cette sémantique dénotationnelle, les limitations de la sémantique opérationnelle.

Cette construction repose sur l'observation des programmes, telle que nous l'avons définie précédemment, mais en réalisant une simplification importante : seuls certains contextes sont retenus pour l'expérimentation. L'observation d'un programme se construit maintenant récursivement de la manière suivante : on considère le résultat de l'exécution du programme, et si c'est une donnée simple (un nombre par exemple), l'observation est terminée, si-

[2]Cette description s'applique aux modèles dénotationnels de la première génération, ceux de la seconde associant aussi une vue « intensionnelle ». Lire par exemple l'introduction de [61], où Ong retrace cette évolution en détail.

non, le résultat est une fonction, et on observe les résultats qu'elle donne lorsqu'on l'applique à tout argument possible.
Rentrons maintenant dans les détails de notre démarche, en la situant.

Pourquoi le λ-calcul paresseux Avant de commencer, justifions rapidement le choix du λ-calcul paresseux[3]. Que le λ-calcul permette de modéliser les aspects fonctionnels des langages de programmation, nul n'en doute. C'est un fait cependant que les implémentations de langages fonctionnels ne réduisent pas sous les abstractions, fait déjà remarqué par Plotkin dans [66, p. 126] ; des raisons d'efficacité dictent ce choix et ne semblent pas devoir être écartées, aujourd'hui comme hier. Pratiquement, c'est donc toujours le λ-calcul paresseux qui est utilisé pour modéliser les langages de programmation. Un pas supplémentaire peut être franchi, d'après Abramsky et Ong (cf. [4] et [5] par exemple) : supplanter la théorie classique du λ-calcul, telle qu'on la trouve dans l'ouvrage de référence de Barendregt [11], par la théorie de la variante paresseuse, où la réduction n'est pas autorisée sous les abstractions. On pourra lire à ce sujet l'introduction éclairante de [4].
Enfin, un mot sur le choix de l'appel par valeur : c'est seulement que nous souhaitons modéliser des langages comme le langage fonctionnel ML, ou comme le langage orienté objets Java, qui utilisent ce mode d'appel. Il est en effet mieux adapté à la manipulation d'objets en mémoire, ou plus généralement à la prise en compte d'effets opérationnels[4] séquentiels[5].

Sémantiques opérationnelles - Les méthodes Le λ-calcul, créé par Church[6] dans les années trente, est un langage permettant de représenter des fonctions et leur utilisation ; sa théorie décrit le quotient de ce langage par une congruence, la β-conversion, modélisation de l'application d'une fonction à un argument. Une sémantique opérationnelle se déduit simplement de la manière suivante : les états de la machine abstraite sont les programmes du langage, et les transitions s'obtiennent à partir de la conversion, en l'orientant. La β-réduction, qui en résulte, transforme l'application

[3] Une présentation syntaxique du λ-calcul se trouve à la fin de cette introduction (p. 113).

[4] On entend par effet opérationnel toute action susceptible de modifier l'état de la machine abstraite sur laquelle s'exécutent les programmes. Ainsi, un langage de programmation peut avoir des effets sur la mémoire, ou sur les entrées et les sorties, ou encore des effets non-déterministes.

[5] Cette affirmation doit être nuancée puisque pour les langages fonctionnels avec appel par nom, les monades ont été utilisées avec succès pour représenter les effets opérationnels.

[6] Pour un survol de son histoire et de ses applications, l'article de Barendregt [12] est parfait, amusant par ses anecdotes et intéressant techniquement.

à un argument d'une fonction en son corps, après remplacement du paramètre formel par l'argument. Muni de cette sémantique opérationnelle, ce langage rudimentaire de programmation a été utilisé pour définir différentes fonctions (par Kleene en particulier), pour finalement aboutir à la thèse de Church : toute fonction calculable est définissable dans le λ-calcul. Trente ans plus tard, avec le développement des ordinateurs, les premières implémentations du λ-calcul paresseux sont envisagées. Ainsi le langage ISWIM développé par Landin (« If You See What I Mean », cf. [44] et [45]), pionnier dans ce domaine, présente une sémantique opérationnelle fondée sur une machine abstraite, la SECD (« Stack, Environment, Control, Dump »). Plotkin, dans son article [66], étudie la correspondance entre les deux sémantiques : à cette fin, il simplifie la sémantique fondée sur la machine abstraite, la SECD, pour en donner une version abstraite, définissant récursivement la même fonction d'évaluation. C'est un des actes fondateurs de la sémantique opérationnelle structurée (par les constructions syntaxiques du langage), qui permet d'éviter toute référence à des machines abstraites de trop bas niveau, sauf à vouloir étudier l'implémentation[7]. Une méthode générale de présentation lui sera bientôt associée, la sémantique naturelle, présentée par Kahn dans [42], un formalisme permettant de former des jugements portant sur des termes, et de définir des règles d'inférence dirigées par la syntaxe.

Avant de poursuivre, fixons la terminologie. Les deux sémantiques opérationnelles évoquées sont structurées, par la relation de réduction d'une part, par la syntaxe d'autre part. C'est donc par la manière dont elles calculent l'interprétation d'un programme qu'elles se distinguent :

- la première procède par *récriture* du programme, suivant la relation de réduction,
- la seconde par une *décomposition* récursive du programme.

Par la suite, nous parlons ainsi de sémantique opérationnelle par *récriture* et par *décomposition*.

Dans [66, Th. 4], Plotkin montre que les sémantiques par récriture et par décomposition sont équivalentes. Cependant, leur pratique s'avère très différente :

- avec la sémantique par récriture, l'exécution est décrite pas à pas (« small step ») et on raisonne par récurrence sur la suite des réductions, aussi bien finie qu'infinie, suivant que le programme termine ou non,

[7] Noter que la sémantique opérationnelle structurée peut formaliser aussi bien une fonction d'évaluation qu'une relation de réduction. En particulier, dans son exposé synthétique de cette méthode [68], Plotkin axiomatise la relation de réduction. Les deux idées clés sont donc la structuration par la syntaxe et l'abstraction.

– avec la sémantique par décomposition, l'exécution est décrite en un pas (« big step ») lorsque le programme termine, et n'est pas décrite sinon, et on raisonne par récurrence structurelle sur la preuve de l'évaluation du programme.

Avec une sémantique par décomposition, peut-on aussi considérer les programmes ne terminant pas ? Telle est la question à laquelle nous répondons dans la première partie, comme illustration de l'approche de l'induction et de la co-induction fondée sur les preuves, développée au premier chapitre. Cette préoccupation n'est pas neuve. C'est dans l'article des Cousot [24] qu'elle semble exprimée et résolue pour la première fois. Une généralisation de la sémantique opérationnelle structurée y est proposée, qui prend en compte la divergence. Le système d'inférence définissant la relation d'évaluation est interprété d'une manière originale, plus qu'inductive, moins que co-inductive, et pour cette raison dite mixte : la part inductive décrit l'évaluation des programmes convergents, la part co-inductive décrivant celle des programmes divergents. Chaque règle d'inférence peut être qualifiée de positive, négative ou neutre, suivant qu'elle peut être utilisée dans une preuve bien fondée, non fondée ou indifféremment fondée ou non. C'est une variante que nous utilisons, qui repose sur la même idée de contrôler l'usage des règles suivant la profondeur des preuves.

Est-ce suffisant ? Non, car il manque une technique pour raisonner avec les preuves infinies, comme on raisonne par récurrence sur une suite de réductions. Cette question est abordée dans l'article d'Ibraheem et Schmidt [39]. La solution proposée revient à parcourir la preuve de l'évaluation suivant l'ordre d'exécution et à raisonner par récurrence sur le parcours. Sa nature algorithmique tient sans doute à l'utilisation envisagée par Schmidt, l'interprétation abstraite (cf. [76]). De notre côté, nous apportons une caractérisation des propriétés ainsi montrées, les propriétés finitaires stables.

Précisons donc notre démarche :

– dans un premier temps, on montre qu'une sémantique par décomposition peut décrire l'exécution d'un programme sous forme de traces, finies ou infinies, tout comme la sémantique par récriture ;
– dans un second temps, pour les deux sémantiques,
  – on caractérise les propriétés finitaires stables (pour un système d'inférence déduit de celui qui définit l'interprétation sémantique), qui sont des propriétés portant sur les programmes et les pas de l'exécution,
  – on montre que ces propriétés sont vérifiées universellement,
  – puis on utilise cette méthode de raisonnement pour conclure à l'équivalence des deux sémantiques ;

– finalement, on réalise une abstraction des sémantiques précédentes, transformant les traces en leur résultat final, une valeur ou la divergence, ce qui permet en particulier d'étendre la sémantique par décomposition classique aux programmes divergents.

Pour conclure, on compare les deux sémantiques opérationnelles, d'un point de vue pratique, celui de leur usage.

Sémantique dénotationnelle - Le modèle des termes

Pour déduire une sémantique dénotationnelle, nous partons de la seule fonction d'évaluation opérationnelle, qui définit pour chaque programme le résultat final de l'exécution, ou bien une valeur (c'est-à-dire une fonction) ou bien la divergence.

Notre objectif est de parvenir à ce que deux programmes aient la même dénotation si et seulement s'ils mènent aux mêmes observations. Que peut-on observer d'un programme du λ-calcul paresseux ? Comme nous l'avons vu plus haut, une manière simple et complète d'y répondre est d'observer les résultats qu'il donne dans chacune de ses utilisations. Utiliser un programme revient à l'intégrer dans un ensemble plus vaste, le contexte d'utilisation. Observer un résultat d'exécution se réduit à déterminer si le programme converge ou non, c'est-à-dire termine ou non. On est donc conduit à définir ainsi l'équivalence de deux programmes :

> deux programmes sont équivalents si dans tout contexte d'utilisation, ou bien ils convergent tous deux, ou bien ils divergent tous deux.

Il est possible de remplacer dans cette définition les programmes par des parties de programme, précisément des termes, à condition de limiter les contextes d'utilisation à ceux qui transforment les termes comparés en programmes. Aussi, il est possible de considérer un pré-ordre plutôt qu'une équivalence, en mesurant à l'aune de la convergence :

> un programme e est inférieur à un autre e' si dans tout contexte d'utilisation, lorsque e termine, alors e' aussi.

La relation d'équivalence s'obtient alors par symétrisation du pré-ordre. Le pré-ordre et l'équivalence sont dits contextuels[8]. Dans cette introduction,

[8]On aurait pu employer le qualificatif « observationnel », fréquemment utilisé et parfaitement adapté, mais ambigu dans notre étude puisque nous allons définir à partir des observations une sémantique observationnelle, dont l'équivalence naturellement associée sera dite « observationnelle » ; on rencontre aussi le qualificatif « opérationnel », trop vague à notre avis, et nous préférons définir l'équivalence opérationnelle par le fait de s'évaluer en la même valeur, autrement dit par ce qu'on appelle généralement la « convertibilité

on s'intéresse principalement à la relation d'équivalence et aux programmes, dans l'unique but de simplifier la présentation, mais il faut se rappeler que les définitions et propriétés mentionnées s'étendent naturellement au préordre et aux termes, comme le montrera notre traitement de la question par la suite.
Reformulons notre objectif :

> que deux programmes aient la même dénotation si et seulement s'ils sont équivalents contextuellement.

Comme l'équivalence contextuelle est une congruence, ce qui signifie que les constructeurs syntaxiques sont compatibles avec elle, on peut essayer de résoudre le problème posé en quotientant l'ensemble des termes par cette congruence et en formant ainsi ce qu'on appelle le *modèle des termes* (modulo cette congruence)[9] ; la question est alors de montrer que la surjection canonique associant à un terme sa classe constitue bien une sémantique dénotationnelle.

Définir une sémantique dénotationnelle Comment se définit une sémantique dénotationnelle ?
Nous suivons la proposition de Henkin, présentée dans son célèbre article de 1950 [35], où il démontre la complétude de la logique d'ordre supérieur ; Henkin introduit une notion de modèle général pour le λ-calcul typé (de la théorie des types de Church, entendue comme une représentation syntaxique de la logique d'ordre supérieur) ; la particularité d'un tel modèle est d'interpréter un type fonctionnel $A \to B$ par un sous-ensemble de l'ensemble des applications de l'interprétation de A dans l'interprétation de B. Cette idée nous est particulièrement utile ici, dans la mesure où un modèle standard du λ-calcul non typé aurait pour domaine un ensemble X vérifiant, à un isomorphisme près, une équation voisine[10] de l'équation

$$X = X \to X\,,$$

où l'ensemble $X \to X$ est interprété de manière standard, c'est-à-dire comme l'ensemble des applications de X dans lui-même. C'est qu'en effet le λ-calcul non typé peut être considéré comme un langage typé, possédant un seul type vérifiant justement l'équation précédente. Bien sûr, pour des raisons de cardinalité, aucun ensemble ne vérifie cette équation.

opérationnelle ».

[9] « Term model » en anglais.

[10] À chaque mode d'appel, son équation.

Nous utilisons aussi une formulation du premier ordre, au sens suivant : bien que les objets considérés représentent des fonctions, nous ne les considérons pas comme telles, mais introduisons une opération, l'application fonctionnelle.
L'approche que nous suivons, classique (voir par exemple la présentation par Mitchell des modèles du λ-calcul simplement typé [57]), est la plus ancienne ; une autre approche, plus récente, existe, et émane de la logique catégorique : l'interprétation s'effectue alors dans une catégorie possédant une structure particulière, précisément une catégorie cartésienne fermée.
De notre point de vue, une sémantique dénotationnelle est simplement une interprétation : elle se définit par un ensemble d'interprétation et une fonction d'interprétation vérifiant certaines propriétés.
Comme un élément d'interprétation représente ou bien une fonction, ou bien la divergence, le domaine forme une structure applicative, une algèbre possédant une seule opération binaire, l'application et possède un élément particulier, noté traditionnellement $\perp$ et représentant la divergence. Toute interprétation de l'application doit vérifier une condition associée à la divergence : être stricte en ses deux arguments, autrement dit, propager la divergence. Comme les éléments d'interprétation différents de $\perp$ représentent des fonctions, une propriété d'extensionalité est vérifiée : un élément fonctionnel est entièrement déterminée par son application à chacun des éléments.
L'interprétation d'un terme associe à tout environnement un élément du domaine d'interprétation : c'est la dénotation du terme dans l'environnement considéré. Elle est compositionnelle en ce sens : dans un environnement donné,

- la dénotation d'une variable est définie par l'environnement, fonction donnant leur valeur aux variables,
- la dénotation d'une abstraction est définie à partir de l'interprétation de son corps,
- la dénotation de l'application entre termes est l'application (interprétée) entre les dénotations des termes.

Le rapport entre les sémantiques opérationnelle et dénotationnelle peut être nuancé ainsi :

- la sémantique dénotationnelle est adéquate lorsque deux termes ayant même interprétation sont équivalents contextuellement,
- la sémantique dénotationnelle est complètement adéquate[11] lorsqu'elle est adéquate et lorsque deux termes équivalents contextuellement ont même interprétation, ce qui constitue le but que nous nous sommes

[11] On traduit ici l'expression anglaise « fully abstract ».

fixé.
Il est au moins demandé à une sémantique dénotationnelle d'être adéquate, sinon à quoi servirait-elle ? On verra par la suite qu'une condition suffisante est de préserver la divergence, soit d'interpréter les programmes divergents par l'élément représentant la divergence, $\bot$.
Revenons au modèle des termes. Comme l'équivalence contextuelle est une congruence, on peut définir l'interprétation de l'application par passage au quotient de l'application entre termes. L'adéquation complète résulte de la définition du quotient, et la compositionalité se montre facilement. Seule la propriété d'extensionalité semble poser problème ; effectivement, examinons les nombreux travaux qui lui ont été consacrés.

La question de l'extensionalité C'est à Plotkin (cf. [67]) et Milner (cf. [53]) qu'on doit la première comparaison entre la sémantique dénotationnelle et l'équivalence contextuelle, pour une extension typée du λ-calcul paresseux, le langage PCF (« Programming Language for Computable Functions »)[12]. Ces deux études viennent après les travaux fondateurs de la théorie des domaines, menés par Scott (cf. [77]), et qui ont permis l'extraordinaire développement de la sémantique dénotationnelle des langages de programmation.
D'un côté, Plotkin montre que le modèle de Scott n'est pas complètement adéquat : la continuité, propriété fondamentale des fonctions dans ce modèle, est trop généreuse, en admettant des fonctions non séquentielles, comme l'alternative dans une évaluation parallèle (« parallel or »), qui permettent de discriminer des programmes pourtant équivalents contextuellement.
De l'autre, en réponse à Plotkin, Milner construit à partir du modèle des termes quotientés par l'équivalence contextuelle, un modèle à la Scott complètement adéquat. Lors de l'étude du modèle des termes, il montre un lemme, dit « lemme des contextes »[13], qui implique la propriété d'extensionalité, sa démonstration se faisant par récurrence sur la longueur des traces définies par la relation de réduction.
Des variations de ce lemme pour l'extensionalité, démontrées de la même façon, se retrouvent dans de nombreux travaux concernant le λ-calcul ; on peut citer par exemple l'étude de Berry [14], concernant les modèles du λ-calcul, ou encore relatifs à sa version paresseuse, les articles d'Abramsky et Ong [5, Prop. 2.2.4] pour l'appel par nom, de Honsell et Lenisa [36, Lem. 5,8,10,12] pour différentes stratégies de réduction, de Mason et al. [52, Th.

[12] Les développements ultérieurs sont décrits par Ong dans l'introduction de [61], ainsi que dans l'introduction de la version rénovée [38], utilisant les jeux et écrite en collaboration avec Hyland.
[13] « Context Lemma » en version originale.

3.5] pour l'appel par valeur.
Une généralisation de ce lemme a été proposée par Howe, dans [37], et c'est elle que nous détaillons maintenant. Par une formalisation plus abstraite, elle permet de paramétrer le lemme par la fonction d'évaluation opérationnelle et de définir des conditions simples pour l'appliquer. Elle repose sur la définition d'une relation de bisimilarité associée à l'évaluation, généralisant la bisimilarité applicative d'Abramsky dans [4]. Cette dernière relation exprime l'équivalence extensionnelle de deux programmes : ils sont équivalents s'ils se comportent de la même façon, du point de vue de la convergence, lorsqu'on leur applique une suite quelconque (éventuellement vide) d'arguments. Pour obtenir l'extensionalité du modèle des termes, il suffit de montrer que les équivalences extensionnelle et contextuelle sont égales. En effet, si c'est le cas, étant donné deux programmes convergents, dont les applications à tout argument sont équivalentes contextuellement, et donc extensionnellement, on peut déduire qu'ils sont équivalents extensionnellement par définition, et donc contextuellement. Montrer l'égalité des équivalences revient à montrer que l'équivalence entre programmes se préserve lorsqu'on étend l'ensemble des contextes considérés, d'où le nom de « lemme des contextes » : purement applicatifs pour l'équivalence extensionnelle, ils deviennent quelconques pour l'équivalence contextuelle. Il suffit pour assurer ce passage de montrer que l'équivalence extensionnelle est une congruence : c'est justement ce qu'apporte la méthode de Howe. Esquissons cet argument que nous retrouverons. Si deux programmes sont équivalents extensionnellement, alors par congruence, ils le sont aussi dans tout contexte d'utilisation, ce qui implique que dans tout contexte, ils convergent tous deux, ou divergent tous deux : ils sont donc équivalents contextuellement.
Suivant une tendance bien affirmée (lire par exemple l'introduction de Gordon et Pitts [32] à l'ouvrage collectif concernant principalement de telles techniques opérationnelles), nous utilisons au cœur de notre démarche la méthode de Howe. Voici comment nous procédons.

1. Nous définissons l'observation d'un programme, en utilisant les seuls contextes applicatifs, autrement dit les contextes appliquant la place à une suite d'arguments : compte tenu de la définition donnée plus haut de l'observation, il vient que la relation d'équivalence entre programmes canoniquement associée à l'observation, une relation de bisimilarité, au sens applicatif, est égale à l'équivalence extensionnelle.

2. Par la méthode de Howe, on montre que la relation de bisimilarité égale l'équivalence contextuelle ; ainsi, les observations permettent de représenter les classes d'équivalence du modèle des termes.

3. À partir des observations utilisées comme dénotations, on définit une sémantique dénotationnelle complètement adéquate du langage, appelée par la suite sémantique observationnelle.

Excepté l'utilisation des observations, la présentation adoptée pour cette sémantique dénotationnelle est proche de celle de Mason, Smith et Talcott dans [52], développée elle-aussi pour des raisons analogues aux nôtres.
Après cette construction, suivant un mouvement habituel, nous cherchons à donner une définition axiomatique de la sémantique observationnelle, c'est-à-dire un ensemble de conditions vérifiées par l'interprétation, la déterminant complètement, à un isomorphisme près. Il s'avère qu'un jeu réduit de conditions suffit :

- la compositionalité, permettant de définir par récurrence structurelle sur les termes leur interprétation,
- l'adéquation avec la sémantique opérationnelle, qui se réduit à la préservation de la divergence, comme on l'a déjà dit,
- une propriété d'extensionalité forte, exprimant que l'égalité (ou l'ordre) sur l'ensemble d'interprétation est définie co-inductivement à partir de la règle d'inférence définissant l'extensionalité, règle qui stipule que deux éléments fonctionnels sont égaux si leur application à tout élément d'interprétation donne le même résultat,
- une propriété d'accessibilité, exprimant que tout élément d'interprétation dénote (une valeur).

Les trois premières propriétés reprennent la définition d'une sémantique dénotationnelle ; seule modification, l'extensionalité forte, qui renforce la propriété originelle, qui affirme seulement que l'égalité est stable pour la règle d'inférence définissant l'extensionalité. Dans le cas des observations, l'extensionalité forte correspond à une formulation co-inductive de l'égalité (ou de l'ordre d'approximation) entre arbres observationnels. Plus généralement, elle exprime la relation entre les équivalences extensionnelle et contextuelle, qu'établit le lemme des contextes.
Ces trois premiers axiomes sont équivalents à ceux des λ-systèmes de transition[14] adéquats, présentés dans l'article déjà cité d'Abramsky et Ong [5].
Quant à l'accessibilité, elle correspond évidemment au fait qu'on considère un modèle de termes. C'est une notion brute de la calculabilité, sans aucune description qualitative.
Avec cette axiomatisation, on semble éloigné de l'élégante caractérisation par des critères de convergence et d'approximation que fournit une inter-

[14] Traduction de « lambda transition systems ».

prétation dans un domaine continu ou algébrique[15]. Cependant, à partir du modèle des termes, il est possible de construire une interprétation dans un domaine algébrique, par complétion, comme le montrent Milner dans [53] et Mason et al. dans [52], pour deux extensions différentes du λ-calcul. De plus, ces modèles vérifient la propriété d'extensionalité forte (cf. [53, Context Lemma] et [52, Th. 4.6]). Comme le montre Abramsky dans son étude du λ-calcul paresseux avec appel par nom, il en est de même de la solution canonique de l'équation récursive entre domaines $D = (D \to D)_\perp$ (cf. [4, Th. 4.1][16]).

Juger de l'utilité Partant de l'axiomatisation de la sémantique observationnelle, on étudie la théorie équationnelle (et inéquationnelle) induite par la sémantique observationnelle sur les termes : deux termes sont équivalents s'ils ont même dénotation dans tout environnement dénotationnel. Comme la sémantique est complètement adéquate, la théorie est évidemment maximale. Ce qui nous intéresse, c'est la manière dont on déduit quelques règles d'inférence classiques, à partir de l'axiomatisation précédente. C'est un bon point de départ pour juger de l'intérêt de la démarche.
Il apparaît que la plupart des règles se déduisent avec une grande facilité des axiomes et de propriétés élémentaires de la sémantique opérationnelle, précisément de la fonction d'évaluation. Un test probant est constitué de la règle permettant le calcul par itération du plus petit point fixe d'un programme. Dans ce cas, la règle est montrée en établissant une propriété de l'équivalence contextuelle, puis par complète adéquation, le résultat vaut aussi pour l'équivalence dénotationnelle. La technique opérationnelle utilisée repose sur l'étude de la réduction d'un programme dans un contexte. Nous formalisons cet argument classique, présent par exemple dans l'article de Mason et al. [52, Th. 3.5], en généralisant aux contextes la décomposition d'un programme, soit en une valeur, soit en un radical dans un contexte de réduction.

Présentation syntaxique du λ-calcul Le lecteur trouvera toutes les définitions suivantes dans l'ouvrage de référence de Barendregt [11].

[15] Un domaine continu est un ensemble ordonné complet (pour ses parties dirigées) et possédant une base ; cela signifie que tout élément est la borne supérieure d'une partie de la base. Un domaine continu est algébrique s'il possède une base formée d'éléments compacts ; un élément est compact si lorsqu'il est inférieur à la borne supérieure d'une partie dirigée, alors il appartient à l'idéal d'ordre engendré par cette partie.

[16] Dans [65], Pitts généralise ce résultat à tout domaine défini récursivement.

Rappelons la grammaire définissant les expressions du λ-calcul :

$$e \quad ::= \quad x \mid \lambda x\, e \mid e\, e\,,$$

où x parcourt un ensemble infini de variables, noté $\mathbf{X}$, $\lambda x\, e$ représente une abstraction fonctionnelle de paramètre x et de corps e, et $e_1\, e_2$ représente l'application de e_1 à l'argument e_2. Les variables libres et liées sont définies de manière habituelle, tout comme l'α-conversion. Les termes sont les classes d'expressions suivant cette dernière relation d'équivalence, qui est une congruence. Toutes les définitions sur les expressions doivent passer au quotient, puisque seuls les termes nous intéressent. C'est le cas par exemple des substitutions. Une substitution est une fonction de l'ensemble des variables dans l'ensemble des termes. L'application de la substitution σ à un terme e, notée $e[\sigma]$, substitue dans e à toute variable libre x de e le terme $\sigma(x)$. Au niveau des expressions, on doit éviter la capture des variables libres de $\sigma(x)$ par les lieurs $\lambda y \ldots$ des abstractions, en renommant le cas échéant les variables liées (soit y). Si la substitution σ a pour domaine $\{x_1, \ldots, x_n\}$, le terme $e[\sigma]$ se note aussi $e[\sigma(x_1)/x_1, \ldots, \sigma(x_n)/x_n]$[17].
Remarquons qu'il n'existe pas de représentation simple des termes. La notation de de Bruijn [25], par exemple, qui permet d'éviter l'α-conversion, complique l'expression des substitutions. Par la suite, on utilise les expressions et réalise le passage des expressions aux termes tacitement.
On notera Λ l'ensemble des termes, Λ^0 l'ensemble des termes clos, c'est-à-dire sans variables libres, appelés aussi programmes. Pour un terme e, on note $\mathbf{FV}(e)$ l'ensemble de ses variables libres.

2.1 Décomposition et récriture

Deux sémantiques opérationnelles sont définies, par décomposition et par récriture. Il s'avère qu'elles sont équivalentes. Une démarche similaire est adoptée pour leur étude, ce qui facilite leur comparaison :

- on décrit l'aspect opérationnel avec un maximum de détails : les deux sémantiques permettent en effet d'associer à chaque programme sa trace d'exécution ;
- comme les traces peuvent être infinies lorsque le programme ne termine pas, on développe pour chaque sémantique une méthode pour raisonner avec les programmes ne terminant pas ;
- ces méthodes sont utilisées pour comparer les deux sémantiques, et montrer leur équivalence ;

[17] Cette notation n'est pas celle de Barendregt dans [11], qui utilise $e[x_1 := \sigma(x_1), \ldots]$.

– pour finir, on fait abstraction des détails opérationnels intermédiaires, pour ne retenir que le résultat de l'exécution.

Préalablement, précisons quelques définitions et notations concernant les traces.

Une *trace* est une suite finie ou infinie de programmes, les programmes formant les configurations sémantiques. Si t est une trace, on note $\mathrm{dom}\, t$ son support, $|t|$ sa longueur (c'est-à-dire le cardinal de son support, ω s'il est infini). Pour tout entier i, si i appartient au support de t, alors $t(i)$ est le $(i+1)^{\text{ième}}$ programme de la trace, sinon $t(i)$ est la valeur de non-définition $\perp$. Si t_1 est une trace finie, t_2 une trace finie ou infinie, la concaténation de t_1 et de t_2 est notée $t_1 \to t_2$; elle admet pour élément neutre la trace vide, notée ε. On décompose toute trace non vide t en un préfixe $\mathrm{p}(t)$ et un programme de fin $\bar{t}$. Si t est finie, alors $\mathrm{p}(t) = t(0) \to \ldots \to t(|t|-2)$, éventuellement vide si t a pour longueur un, et $\bar{t} = t(|t|-1)$. Si t est infinie, alors $\mathrm{p}(t) = t$ et $\bar{t} = \perp$. Enfin, si e est un programme, on note $< t\, e >$ la trace $(t(0)\, e) \to \ldots (t(n)\, e)[\ldots]$, et symétriquement, $< e\, t >$ la trace $(e\, t(0)) \to \ldots (e\, t(n))[\ldots]$.

Par la suite, l'ensemble des traces finies non vides est noté $(\Lambda^0)^+$, et celui des traces infinies, $(\Lambda^0)^\omega$.

Les traces infinies peuvent être représentées par des arbres respectant une certaine signature. Précisément, on utilise l'isomorphisme entre l'ensemble des traces infinies et l'ensemble des arbres respectant la signature à une seule sorte, dont les étiquettes sont les programmes et ont pour arité un. Plus généralement, toute trace peut être représentée par un arbre : il suffit d'ajouter à la signature précédente une étiquette ε, d'arité nulle, et représentant la trace vide, pour obtenir un isomorphisme entre l'ensemble des traces et l'ensemble des arbres respectant cette signature.

2.1.1 Décomposer pour exécuter

Commençons par donner une sémantique par décomposition du λ-calcul paresseux, avec appel par valeur. Fidèle au programme annoncé, on s'intéresse aux traces d'exécution des programmes, et non seulement aux résultats qu'ils donnent.

À chaque programme, on associe donc une suite maximale d'états sémantiques, ou configurations, décrivant l'exécution du programme. Cette sémantique du λ-calcul se définit à partir d'un système d'inférence, portant sur des jugements de la forme $e \Downarrow t$, où e est un programme et t une trace, résultat de l'exécution de e.

Dans sa forme habituelle, une telle sémantique ne rend pas compte de la

$$\frac{\emptyset}{\lambda x\, e \Downarrow \lambda x\, e}\ [\text{ABS}]$$

$$\frac{+\ e_1 \Downarrow t_1 \quad +\ e_2 \Downarrow t_2 \quad +\ e[v/x] \Downarrow t}{e_1\, e_2 \Downarrow < \mathrm{p}(t_1)\, e_2 > \rightarrow < \overline{t_1}\, t_2 > \rightarrow t}\ [\text{APP+}] \quad \left(\begin{array}{l} t_1, t_2, t \in (\Lambda^0)^+ \\ \overline{t_1} = \lambda x\, e, \overline{t_2} = v \end{array} \right)$$

$$\frac{-\ e_1 \Downarrow t_1}{e_1\, e_2 \Downarrow < t_1\, e_2 >}\ [\text{APP1-}] \quad (t_1 \in (\Lambda^0)^\omega)$$

$$\frac{+\ e_1 \Downarrow t_1 \quad -\ e_2 \Downarrow t_2}{e_1\, e_2 \Downarrow < \mathrm{p}(t_1)\, e_2 > \rightarrow < \overline{t_1}\, t_2 >}\ [\text{APP2-}] \quad \left(\begin{array}{l} t_1 \in (\Lambda^0)^+ \\ t_2 \in (\Lambda^0)^\omega \end{array} \right)$$

$$\frac{+\ e_1 \Downarrow t_1 \quad +\ e_2 \Downarrow t_2 \quad -\ e[v/x] \Downarrow t}{e_1\, e_2 \Downarrow < \mathrm{p}(t_1)\, e_2 > \rightarrow < \overline{t_1}\, t_2 > \rightarrow t}\ [\text{APP3-}] \quad \left(\begin{array}{l} t_1, t_2 \in (\Lambda^0)^+ \\ t \in (\Lambda^0)^\omega \\ \overline{t_1} = \lambda x\, e, \overline{t_2} = v \end{array} \right)$$

Tab. 2.1 – Sémantique par décomposition : traces d'exécution

divergence puisqu'elle repose sur une définition inductive, n'autorisant que des preuves finies. Pour évaluer aussi bien les programmes divergents que ceux convergents, il est nécessaire de recourir à une interprétation mixte du système d'inférence, en partie inductive, en partie co-inductive (cf. p. 75 pour la définition et les notations). Rappelons que dans cette interprétation, chaque prémisse d'une règle se voit qualifiée de « positive », « négative » ou « neutre » ; dans la preuve d'un jugement, une règle est utilisable si chacune de ses prémisses, suivant qu'elle est positive, négative ou neutre, est la conclusion d'une preuve respectivement fondée, non fondée ou quelconque.

La relation d'évaluation donnant la trace est définie par le système d'inférence de la table 2.1 (p. 116) ; ses jugements sont de la forme $e \Downarrow t$, où e est un programme et t une trace. On observe que les prémisses positives concernent les traces finies, les négatives, les traces infinies. Remarquons aussi qu'une preuve finie n'utilise que les règles [ABS] et [APP+], et qu'une preuve infinie ne peut se conclure que par une des règles [APP1-], [APP2-] ou [APP3-]. Ainsi, les conclusions des preuves finies portent sur des traces finies, celles des preuves infinies sur des traces infinies.
Avant d'établir les propriétés de cette relation d'évaluation, traitons deux exemples, pour bien montrer la différence entre les prémisses positives et

négatives. Notons I la fonction identité, $\lambda x\, x$, et δ le programme d'auto-application, $\lambda x\, x\, x$. Donnons la preuve que $I\, I$ a pour trace $(I\, I) \to I$:

$$\dfrac{\dfrac{\emptyset}{I \Downarrow I}\,[\text{ABS}] \quad \dfrac{\emptyset}{I \Downarrow I}\,[\text{ABS}] \quad \dfrac{\emptyset}{I \Downarrow I}\,[\text{ABS}]}{I\, I \Downarrow I\, I \to I}\,[\text{APP+}] \quad .$$

Cette preuve est finie, tout comme sa trace. À l'opposé, le programme $\delta\, \delta$ ne termine pas. Voici donc une preuve infinie que sa trace est $\delta\, \delta \to \ldots \to \delta\, \delta \ldots$, trace abrégée ci-dessous en $(\delta\, \delta)^\omega$:

$$\dfrac{\dfrac{\emptyset}{\delta \Downarrow \delta}\,[\text{ABS}] \quad \dfrac{\emptyset}{\delta \Downarrow \delta}\,[\text{ABS}] \quad \dfrac{\ldots}{\delta\, \delta \Downarrow (\delta\, \delta)^\omega}\,[\text{APP3-}]}{\delta\, \delta \Downarrow \delta\, \delta \to (\delta\, \delta)^\omega}\,[\text{APP3-}] \quad .$$

Le système d'inférence définit une relation d'évaluation dont la principale propriété est la suivante : tout programme s'exécute suivant une unique trace.

Pour montrer cette propriété, on s'intéresse tout d'abord aux programmes convergents, puis à ceux divergents. On dit qu'un programme *converge* (ou termine) s'il possède une trace d'exécution finie, et qu'il *diverge* (ou ne termine pas) s'il en possède une infinie. Le raisonnement suit les étapes suivantes :

1. un programme convergent possède une trace finie unique,
2. un programme convergent ne peut être divergent,
3. un programme non convergent est divergent, et possède une trace infinie unique.

Ainsi, si un programme est convergent, alors il s'exécute suivant une unique trace, qui de plus est finie, sinon, il est divergent, et s'exécute suivant une unique trace, cette fois infinie.

Montrons tout d'abord qu'un programme convergent ne possède qu'une seule trace finie.

2.1.1 Proposition (Convergence déterministe)
Soient e un programme, et t_1 et t_2 deux traces finies. Si le jugement $e \Downarrow t_1$ est valide, tout comme $e \Downarrow t_2$, alors les traces t_1 et t_2 sont égales.

Démonstration
On procède par récurrence structurelle sur la preuve finie de $e \Downarrow t_1$. Pour

les deux règles applicables, [ABS] et [APP+], c'est immédiat.
⌞

On pourra donc parler par la suite pour un programme convergent e de *sa* trace finie, trace qu'on notera $T(e)$.
Montrons maintenant qu'un programme convergent ne diverge pas.

2.1.2 Proposition (Convergence implique non divergence)
Soit e un programme convergent. Alors e n'est pas divergent : pour toute trace infinie t, le jugement $e \Downarrow t$ n'est pas valide.

Démonstration
On procède par récurrence structurelle sur la preuve finie de $e \Downarrow T(e)$.
Pour chaque règle par laquelle $e \Downarrow T(e)$ peut se conclure, on considère une trace infinie t et on raisonne par l'absurde, en supposant que $e \Downarrow t$ soit la conclusion d'une preuve infinie.
• règle [ABS]
Le programme e est une abstraction, et aucune des règles [APP1-], [APP2-] ou [APP3-] n'est applicable.
• règle [APP+]
Les règles [APP1-], [APP2-] ou [APP3-] peuvent conclure $e \Downarrow t$.
Supposons que ce soit la règle [APP1-]. On obtient une contradiction par l'hypothèse de récurrence appliquée à la prémisse gauche de $e \Downarrow T(e)$.
Supposons que ce soit la règle [APP2-]. On obtient une contradiction par l'hypothèse de récurrence appliquée à la prémisse centrale de $e \Downarrow T(e)$.
Supposons que ce soit la règle [APP3-]. Dans ce cas, par la proposition 2.1.1 (p. 117), $e \Downarrow T(e)$ et $e \Downarrow t$ ont les mêmes prémisses gauche et centrale. Les prémisses droites de $e \Downarrow T(e)$ et de $e \Downarrow t$ définissent donc la trace d'un même programme, d'où la contradiction en appliquant l'hypothèse de récurrence à la prémisse droite de $e \Downarrow T(e)$.
⌞

Par la suite, on désignera par **C** l'ensemble des programmes convergents, et **D** l'ensemble complémentaire des programmes non convergents. La proposition suivante montre que **D** est contenu dans l'ensemble des programmes divergents.

2.1.3 Proposition (Non convergence implique divergence)
Tout programme e qui ne converge pas diverge.

Démonstration
Montrons que tout programme qui ne converge pas peut s'exécuter en une

trace infinie. On commence par associer à chaque programme de $\mathbf{D}$ une trace d'exécution infinie, puis on conclut en montrant que chaque programme s'évalue bien en la trace qui lui est associée.
Pour associer à chaque programme de $\mathbf{D}$ une trace infinie, on va résoudre un système d'équations récursives sur l'ensemble des traces infinies. Considérons le système d'inconnues $(X_e)_{e\in\mathbf{D}}$ et formé des équations de la table 2.2. Les trois cas présentés épuisent les possibilités pour un programme non

$$X_{e_1\,e_2} = \begin{cases} < X_{e_1}\, e_2 > & \text{si } e_1 \in \mathbf{D}\,, \quad (2.1) \\ < \mathrm{p}(T(e_1))\, e_2 > \rightarrow < \overline{T(e_1)}\, X_{e_2} > & \text{si } e_1 \in \mathbf{C}, e_2 \in \mathbf{D}\,, \quad (2.2) \\ < \mathrm{p}(T(e_1))\, e_2 > \rightarrow & \\ \quad < \overline{T(e_1)}\, T(e_2) > \rightarrow X_{e[v/x]} & \text{si } \begin{array}{l} e_1, e_2 \in \mathbf{C}\,, \\ e[v/x] \in \mathbf{D}\,, \end{array} \quad (2.3) \\ & \text{avec } \begin{array}{l} \overline{T(e_1)} = \lambda x\, e\,, \\ \overline{T(e_2)} = v\,. \end{array} \end{cases}$$

TAB. 2.2 – Dém. prop. 2.1.3

convergent.
Les applications $< -e >$ et $< e- >$ sont des extensions à l'ensemble des traces d'applications définies initialement sur l'ensemble des programmes. D'après la proposition 1.1.8 (p. 64), ce sont des opérations autorisées dans les équations récursives. Il est donc possible d'appliquer le théorème 1.1.9 (p. 65) si l'on montre que ce système, qui n'est pas gardé à cause des équations de type 2.1 et 2.2 (lorsque e_1 est une abstraction), est équivalent à un système gardé. Montrons donc par l'absurde qu'on ne peut avoir une suite infinie d'équations $(X_{e_i} =< X_{e_{i+1}}\, e'_i >$ ou $< e'_i\, X_{e_{i+1}} >)_{i\in\mathbb{N}}$ de type 2.1 ou 2.2. Si c'était le cas, on obtiendrait une suite infinie strictement décroissante de sous-termes, ce qui est impossible. Après un nombre fini de telles équations, on rencontre nécessairement une équation gardée de type 2.2 ou de type 2.3, et par des expansions successives des inconnues, cette suite devient équivalente à une équation gardée, au sens du théorème 1.1.9 (p. 65). Le système d'équations peut donc se récrire en un système équivalent gardé, admettant une unique solution. Pour e appartenant à $\mathbf{D}$, notons $T(e)$ la valeur de la solution du système en X_e ; par définition, $T(e)$ est une trace

infinie.
À partir des preuves finies $(P(e \Downarrow T(e)))_{e\in\mathbf{C}}$ obtenues pour les programmes convergents et des traces infinies $(T(e))_{e\in\mathbf{D}}$, nous allons construire pour chaque programme e de $\mathbf{D}$ une preuve infinie de son évaluation en $T(e)$, en résolvant un système d'équations récursives, d'inconnues $((Y_e)_{e\in\mathbf{C}}, (X_e)_{e\in\mathbf{D}})$ et présenté dans la table 2.3. Ce système est quasi-uniforme et compatible

$$
\begin{aligned}
X_{e_1\,e_2} \;=\; & \left\{ \begin{array}{ll}
\dfrac{-\ X_{e_1}}{e_1\,e_2 \Downarrow T(e_1\,e_2)} & \text{si } e_1 \in \mathbf{D}\,, \\[2ex]
\dfrac{+\ Y_{e_1} \quad -\ X_{e_2}}{e_1\,e_2 \Downarrow T(e_1\,e_2)} & \text{si } \left\{ \begin{array}{l} e_1 \in \mathbf{C}\,, \\ e_2 \in \mathbf{D}\,, \end{array} \right. \\[2ex]
\dfrac{+\ Y_{e_1} \quad +\ Y_{e_2} \quad -\ X_{e[v/x]}}{e_1\,e_2 \Downarrow T(e_1\,e_2)} & \text{si } \left\{ \begin{array}{l} e_1 \in \mathbf{C}\,, \\ e_2 \in \mathbf{C}\,, \\ e[v/x] \in \mathbf{D}\,, \end{array} \right. \\
& \text{avec } \left\{ \begin{array}{l} \overline{T(e_1)} = \lambda x\, e\,, \\ \overline{T(e_2)} = v\,, \end{array} \right.
\end{array} \right. \\
Y_e \;=\; & P(e \Downarrow T(e)) \quad (e \in \mathbf{C})\,.
\end{aligned}
$$

Tab. 2.3 – Dém. prop. 2.1.3

avec le système d'inférence définissant l'évaluation en traces ; d'après la proposition 1.2.4 (p. 75), pour tout programme e appartenant à $\mathbf{D}$, la valeur de la solution en X_e est une preuve de l'évaluation de e en $T(e)$.
On a donc montré qu'un programme non convergent est divergent.
⌞

Résumons ce qu'on a montré par les trois dernières propositions. Un programme est soit convergent, soit divergent ; s'il est convergent, il possède une unique trace, qui est finie ; s'il est divergent, il peut s'exécuter suivant une ou plusieurs traces, toutes infinies. Tout programme s'exécute donc en au moins une trace, qu'on note $T(e)$.

Pour montrer l'unicité de la trace infinie pour un programme divergent, on va utiliser une technique très générale, qui évite de considérer les preuves infinies, dues à la divergence.

On s'intéresse à une classe particulière de propriétés, définies par des pré-

dicats binaires, ayant pour paramètres un programme et un entier naturel. Considérons une telle propriété définie par un prédicat dont les jugements sont notés $e : n$, pour tout programme e et tout entier naturel n. Typiquement, la propriété concernera le (n+1)$^{\text{ième}}$ programme de la trace du programme e, ou le préfixe de longueur $n+1$: elle est ainsi qualifiée de *finitaire*. Pour certaines propriétés, dites *stables*, il est possible de construire une preuve finie de tout jugement $e : n$, en utilisant une partie finie de la preuve éventuellement infinie de l'évaluation de e. Ces propriétés sont dites stables car leur extension l'est, pour un système d'inférence canoniquement associé à celui qui définit les traces d'exécution.

2.1.4 Définition (**Sémantique par décomposition : Propriété finitaire stable**)

Considérons une propriété finitaire, définie par un prédicat de jugements $(e : n)_{e\in\Lambda^0, n\in\mathbb{N}}$. *Elle est* stable *si l'ensemble des jugements vrais est stable pour le système d'inférence de la table 2.4.*

$$\frac{\emptyset}{\lambda x\, e : n}\,[\text{ABS n}]\,,$$

$$\frac{e_1 : n}{e_1\, e_2 : n}\,[\text{APP1 n}] \quad (n < |T(e_1)| - 1)\,,$$

$$\frac{e_1 : n \quad e_2 : p}{e_1\, e_2 : n}\,[\text{APP1-2 n}] \quad \left(\begin{array}{l} n = |T(e_1)| - 1 \\ p = 0 \end{array}\right),$$

$$\frac{e_2 : p}{e_1\, e_2 : n}\,[\text{APP2 n}] \quad \left(\begin{array}{l} |T(e_1)| \leq n < |T(e_1)| + |T(e_2)| - 1 \\ p = n - (|T(e_1)| - 1) \end{array}\right),$$

$$\frac{e[v/x] : p}{e_1\, e_2 : n}\,[\text{APP3 n}] \quad \left(\begin{array}{l} \overline{|T(e_1)| + |T(e_2)| - 1} \leq n \\ \overline{T(e_1)} = \lambda x\, e, \overline{T(e_2)} = v \\ p = n - (|T(e_1)| + |T(e_2)| - 1) \end{array}\right).$$

Tab. 2.4 – Sémantique par décomposition : propriété finitaire stable

Les règles du système précédent suivent assez naturellement celles du système d'évaluation. Seules la règle [APP1-2 n] présente une particularité, une condition de frontière. Lors de l'évaluation d'une application, la configuration sémantique terminant l'évaluation de la fonction est aussi celle qui

commence l'évaluation de l'argument ; on choisit donc de déduire la propriété concernant cette configuration de l'évaluation de la fonction et de l'évaluation de l'argument.
La stabilité implique que l'ensemble des jugements vrais contient l'ensemble engendré inductivement par le système précédent. Comme il s'avère qu'une preuve finie de tout jugement dans le système précédent peut être construite, on peut conclure que toute propriété finitaire stable est vérifiée universellement.

2.1.5 Proposition (Sémantique par décomposition : Lemme des propriétés finitaires stables)

Considérons une propriété finitaire stable, définie par un prédicat de jugements $(e : n)_{e\in\Lambda^0,n\in\mathbb{N}}$. *Alors elle est vérifiée universellement : pour tout programme* e *et tout entier* n, *on a*

$$e : n\,.$$

Démonstration
Primo, on construit pour chaque programme e et chaque entier n une preuve dans le système d'inférence de la définition 2.1.4 (p. 121), secundo, on montre qu'il n'existe pas de preuves infinies dans ce système, ce qui montre que les preuves précédemment construites sont finies, tertio, on conclut par la stabilité que la propriété est toujours vérifiée.

Définissons un système d'équations récursives, d'inconnues $(X_{e,n})_{e\in\Lambda^0,n\in\mathbb{N}}$, par les équations de la table 2.5. Ce système est quasi-uniforme et compatible avec le système d'inférence de la définition 2.1.4. D'après la proposition 1.2.3 (p. 73), pour tout programme e et tout entier n, la valeur de l'unique solution en $X_{e,n}$ est une preuve de $e : n$ suivant ce système d'inférence.

Montrons maintenant que ce système d'inférence n'admet que des preuves finies. On démontre par récurrence sur l'entier n que toute preuve de conclusion $e : n$, e étant un programme quelconque, est finie.
• $n = 0$
On montre par récurrence sur le programme e que toute preuve de conclusion $e : 0$ est finie.
◦ $e = \lambda x\, b$
Le jugement $e : 0$ est la conclusion de la règle [ABS 0], et constitue donc un axiome.
◦ $e = e_1\, e_2$
Le jugement $e : 0$ est la conclusion de la règle [APP1 0] ou [APP1-2 0]. Dans les deux cas, on peut appliquer l'hypothèse de récurrence aux prémisses, qui concernent dans le premier cas, e_1, dans le second cas, e_1 et e_2, ce qui permet

$$
\begin{aligned}
X_{\lambda x\, e,n} &= \frac{\emptyset}{\lambda x\, e \,:\, n}\,, \\
X_{e_1\, e_2,n} &= \begin{cases}
\dfrac{X_{e_1,n}}{e_1\, e_2 \,:\, n} & \text{si } n < |T(e_1)| - 1\,, \\[2ex]
\dfrac{X_{e_1,n} \quad X_{e_2,0}}{e_1\, e_2 \,:\, n} & \text{si } n = |T(e_1)| - 1\,, \\[2ex]
\dfrac{X_{e_2,p}}{e_1\, e_2 \,:\, n} & \text{si } \begin{array}{l} |T(e_1)| \leq n < |T(e_1)| + |T(e_2)| - 1\,, \\ p = n - (|T(e_1)| - 1)\,, \end{array} \\[2ex]
\dfrac{X_{e[v/x],p}}{e_1\, e_2 \,:\, n} & \text{si } \begin{array}{l} |T(e_1)| + |T(e_2)| - 1 \leq n\,, \\ \overline{T(e_1)} = \lambda x\, e, \overline{T(e_2)} = v\,, \\ p = n - (|T(e_1)| + |T(e_2)| - 1)\,. \end{array}
\end{cases}
\end{aligned}
$$

TAB. 2.5 – Dém. prop. 2.1.5

de conclure.

• $n > 0$

On suppose que pour tout entier p strictement inférieur à n, pour tout programme e, on a $e : p$. On montre par récurrence sur le programme e que toute preuve de conclusion $e : n$ est finie.

◦ $e = \lambda x\, b$

Le jugement $e : n$ est la conclusion de la règle [ABS n], et constitue donc un axiome.

◦ $e = e_1\, e_2$

Si le jugement $e : n$ est la conclusion de la règle [APP1 n] ou [APP1-2 n], chacune de ses prémisses est de la forme $e' : k$, où e' est égal soit à e_1, soit à e_2 ; on applique donc l'hypothèse de récurrence aux sous-termes e_1 et e_2 pour conclure.

Si le jugement $e : n$ est la conclusion de la règle [APP2 n] ou [APP3 n], sa prémisse est de la forme $e' : p$, où p est strictement inférieur à n ; on applique l'hypothèse de récurrence à l'entier p pour conclure.

Finalement, pour tout programme e et tout entier n, le jugement $e : n$, qui admet une preuve finie, appartient à l'ensemble engendré inductivement

par le système de la définition 2.1.4. Comme la propriété est stable, le jugement $e : n$ est donc vrai.
⌞

Avec l'aide de ce lemme fondamental, il est facile de montrer l´unicité de la trace infinie associée à un programme divergent.

2.1.6 Proposition (Divergence déterministe)
Soient e un programme, et t_1 et t_2 deux traces infinies. Si le jugement $e \Downarrow t_1$ est valide, tout comme $e \Downarrow t_2$, alors les traces t_1 et t_2 sont égales.

Démonstration
Considérons la propriété finitaire suivante :

$$e : n \stackrel{def}{\Leftrightarrow} \forall t_1, t_2 \in (\Lambda^0)^\omega \,.\, e \Downarrow t_1 \wedge e \Downarrow t_2 \Rightarrow t_1(n) = t_2(n)\,.$$

Montrons qu'elle est stable. Passons en revue les différentes règles d'inférence. Il s'agit à chaque fois de montrer que si les prémisses sont vérifiées, alors la conclusion aussi.
On utilise sans le mentionner le fait que tout programme convergent s'évalue en une unique trace, $T(e)$.
• [ABS n]
C'est trivialement vérifié, puisqu'on a $\lambda x\, e \Downarrow \lambda x\, e$.
Considérons désormais deux programmes e_1 et e_2, et un entier n.
• [APP1 n]
Supposons $n < |T(e_1)| - 1$ et $e_1 : n$.
Si t est une trace telle que $e_1\, e_2 \Downarrow t$, on a par définition de l'évaluation $t(n) = T(e_1)(n)\, e_2$, et on peut conclure.
• [APP1-2 n]
Supposons $n = |T(e_1)| - 1$, $e_1 : n$ et $e_2 : 0$.
Si t est une trace telle que $e_1\, e_2 \Downarrow t$, on a par définition de l'évaluation $t(n) = T(e_1)(n)\, e_2$, et on peut conclure.
• [APP2 n]
Supposons $|T(e_1)| \leq n < |T(e_1)| + |T(e_2)| - 1$, $e_1 : n$ et $e_2 : p$, avec $p = n - (|T(e_1)| - 1)$.
Étant donné deux traces infinies t_1 et t_2, supposons $e_1\, e_2 \Downarrow t_1$ et $e_1\, e_2 \Downarrow t_2$. Il existe une trace t'_1 telle que $e_1\, e_2 \Downarrow t_1$ admet pour prémisse $e_2 \Downarrow t'_1$, et une trace t'_2 telle que $e_1\, e_2 \Downarrow t_2$ admet pour prémisse $e_2 \Downarrow t'_2$. Il est facile de voir que $t_1(n) = \overline{T(e_1)}\, t'_1(p)$ et $t_2(n) = \overline{T(e_1)}\, t'_2(p)$. De $e_2 : p$, on déduit que $t'_1(p) = t'_2(p)$, puis que $t_1(n) = t_2(n)$.
• [APP3 n]

Supposons $|T(e_1)| + |T(e_2)| - 1 \leq n$ et $e[v/x] : p$, avec $p = n - (|T(e_1)| + |T(e_2)| - 1)$, $\overline{T(e_1)} = \lambda x\, e$ et $\overline{T(e_2)} = v$.
Étant donné deux traces infinies t_1 et t_2, supposons $e_1\, e_2 \Downarrow t_1$ et $e_1\, e_2 \Downarrow t_2$. Il existe une trace t'_1 telle que $e_1\, e_2 \Downarrow t_1$ admet pour prémisse $e[v/x] \Downarrow t'_1$, et une trace t'_2 telle que $e_1\, e_2 \Downarrow t_2$ admet pour prémisse $e[v/x] \Downarrow t'_2$. Il est facile de voir que $t_1(n) = t'_1(p)$ et $t_2(n) = t'_2(p)$. De $e[v/x] : p$, on déduit que $t'_1(p) = t'_2(p)$, puis que $t_1(n) = t_2(n)$.
⌞

En conclusion, tout programme possède une unique trace d'exécution : la sémantique par décomposition proposée rend compte des programmes tant convergents que divergents.

2.1.2 Réduire pour exécuter

Le système définissant les traces d'exécution ne met l'accent ni sur la relation de réduction, ni sur la stratégie d'évaluation. Ce sont les règles [APP+] et [APP3-] (de la table 2.1 (p. 116)) qui induisent une transition entre programmes, correspondant à la β-réduction habituelle

$$(\lambda x\, e)\, v \xrightarrow{\mathrm{r}} e[v/x]\,.$$

Quant à la stratégie de réduction, elle est exprimée par les règles concernant l'application : la fonction est d'abord évaluée, puis l'argument avant son passage. Il est en fait possible d'exprimer la même sémantique, en définissant tout d'abord la relation de réduction et la stratégie d'évaluation, puis les traces maximales pour cette relation.

Pour la relation de réduction, sont importants les programmes ne se réduisant pas, appelés *valeurs* : ce sont les abstractions closes.

$$v \quad ::= \quad \lambda x\, e\,,$$

où $\mathbf{FV}(\lambda x\, e) = \emptyset$. On désignera l'ensemble des valeurs par $\mathbf{V}$.
Un programme, s'il n'est pas une valeur, peut se décomposer en un *radical*, qui va se réduire, et une continuation, représentant ce qui reste à évaluer après que le radical a été réduit.
Les radicaux qui se réduisent dans un programme sont formés par des applications d'abstractions closes à des valeurs.

$$r \quad ::= \quad (\lambda x\, e)\, v\,,$$

où $\mathbf{FV}(\lambda x\, e) = \emptyset$.
Un *contexte de réduction*, représentant la continuation, est un terme clos à

$$\frac{\emptyset}{(\lambda x\, e)\, v \xrightarrow{\mathrm{r}} e[v/x]}\ [\beta]$$

$$\frac{r \xrightarrow{\mathrm{r}} e}{E[r] \xrightarrow{\mathrm{r}} E[e]}\ [\mathrm{RED}]$$

TAB. 2.6 – Relation de réduction

une place, linéaire en sa place : il possède une seule variable libre, la *place*, à l'occurrence de plus unique ; on réserve à ce seul usage une variable, notée $-$[18]. Un contexte de réduction se construit autour de la place, en appliquant des valeurs, ou en passant des arguments non évalués :

$$E \quad ::= \quad - \mid v\, E \mid E\, e\,,$$

où $\mathbf{FV}(e) = \emptyset$.
Tout programme peut se décomposer de manière unique en une valeur ou un radical placé dans un contexte de réduction : si e est un programme, ou bien c'est une valeur, ou bien il existe un unique radical r et un unique contexte E tel que $e = E[r]$ ($E[r]$ abrégeant $E[r/-]$).

La relation de réduction $\xrightarrow{\mathrm{r}}$ est définie (inductivement) par le système d'inférence de la table 2.6 (p. 126). L'axiome exprime la β-réduction, la règle permettant de passer au contexte de réduction. La relation est *déterministe* : si $e \xrightarrow{\mathrm{r}} e_1$ et si $e \xrightarrow{\mathrm{r}} e_2$, alors $e_1 = e_2$. En effet, si le système est restreint à l'axiome, la relation ainsi définie l'est, et de plus, la décomposition en un radical dans un contexte de réduction est unique. Elle est aussi *totale*, dans le sens suivant : tout programme qui n'est pas une valeur se réduit, conformément à la définition informelle que nous avons donnée des valeurs.

Nous nous intéressons maintenant à la construction des traces à partir de la relation de réduction. Cette construction suppose seulement la donnée d'une relation de réduction totale ; l'hypothèse du déterminisme n'est en effet pas nécessaire.

Une fois que la relation de réduction est définie, il est facile d'associer à chaque programme l'ensemble de ses traces d'exécution. On considère des jugements de la forme $e \Downarrow_r t$, exprimant que le programme e peut s'exécuter suivant la trace t. Le système d'inférence de cette nouvelle relation d'évaluation est présenté dans la table 2.7 (p. 127), étant donné une relation de réduction $\xrightarrow{r}$ totale. De l'observation des règles et des preuves, deux

[18] Cela signifie que cette variable ne sert pas à construire de termes autres que les

$$\frac{\emptyset}{v \Downarrow_r v}\,[\text{VAL}] \quad (v \in \mathbf{V})$$

$$\frac{+\,e' \Downarrow_r t}{e \Downarrow_r e \to t}\,[\text{RED+}] \quad \left(\begin{array}{l} e, e' \in \Lambda^0 \\ t \in (\Lambda^0)^+ \\ e \xrightarrow{\text{r}} e' \end{array}\right)$$

$$\frac{-\,e' \Downarrow_r t}{e \Downarrow_r e \to t}\,[\text{RED-}] \quad \left(\begin{array}{l} e, e' \in \Lambda^0 \\ t \in (\Lambda^0)^\omega \\ e \xrightarrow{\text{r}} e' \end{array}\right)$$

Tab. 2.7 – Sémantique par récriture : traces d'exécution

remarques viennent immédiatement :
- les traces finies et infinies sont obtenues respectivement par des preuves finies et infinies,
- une trace commence toujours par le programme à évaluer.

Dans ce système, tout programme s'évalue en au moins une trace.

2.1.7 Proposition (Existence d'une trace)
Soit $\xrightarrow{r}$ une relation de réduction totale. Alors tout programme possède au moins une trace d'exécution suivant le système d'inférence de la table 2.7 (p. 127).

Démonstration
La démonstration se fait en deux temps.
Primo, on associe à chaque programme une trace, en résolvant un système d'équations récursives sur les traces, directement inspiré du système d'inférence de la table 2.7. Comme la relation de réduction est totale, il est possible d'associer à tout programme e qui n'est pas une valeur un programme $r(e)$ tel que e se réduit en $r(e)$. Considérons alors le système d'inconnues $(X_e)_{e\in\Lambda^0}$ et formé des équations suivantes :

$$X_e = \begin{cases} e & \text{si } e \text{ valeur,} \\ e \to X_{r(e)} & \text{sinon.} \end{cases}$$

Pour tout programme e, notons $T_r(e)$ la valeur de la solution en X_e.
Secundo, on montre que tout programme e s'évalue en $T_r(e)$ suivant le système d'inférence de la table 2.7.

contextes, en particulier qu'elle n'est jamais liée.

Tout d'abord, montrons que pour tout programme e, si $T_r(e)$ est une trace finie, alors $e \Downarrow_r T_r(e)$ est un jugement valide. C'est immédiat, en procédant par récurrence sur la longueur des traces.
À tout programme e tel que $T_r(e)$ est une trace infinie, associons une preuve de $e \Downarrow_r T_r(e)$ en résolvant le système d'équations récursives d'inconnues $(X_e)_{e \text{ tel que } |T_r(e)|=\omega}$ et formé des équations suivantes :

$$X_e = \frac{- X_{r(e)}}{e \Downarrow_r T_r(e)}.$$

Ce système est quasi-uniforme et compatible avec le système d'inférence définissant l'évaluation en traces (cf. tab. 2.7). Par la proposition 1.2.4 (p. 75), pour tout programme e tel que la trace $T_r(e)$ est infinie, la valeur de la solution en X_e est une preuve de $e \Downarrow_r T_r(e)$.
⌞

Comme pour le système définissant la sémantique par décomposition, afin de raisonner sur l'évaluation, avec ses preuves éventuellement infinies, on peut définir des propriétés finitaires stables, et le mode de raisonnement associé correspond ici à celui qu'on attend, une récurrence sur la position dans une trace. On note désormais $e :_r n$ ($e \in \Lambda^0, n \in \mathbb{N}$) les jugements de ces propriétés, pour souligner le système auquel se réfère la stabilité.

2.1.8 Définition (Sémantique par récriture : Propriété finitaire stable)

Soit $\xrightarrow{r}$ une relation de réduction totale. Considérons une propriété finitaire, *définie par un prédicat de jugements $(e :_r n)_{e \in \Lambda^0, n \in \mathbb{N}}$. Elle est* stable *si l'ensemble des jugements vrais est stable pour le système d'inférence de la table 2.8.*

$$\frac{\emptyset}{v :_r n}\ [\text{VAL n}] \quad (v \in \mathbf{V}, n \geq 0)\,,$$

$$\frac{\emptyset}{e :_r 0}\ [\text{RED 0}] \quad (e \text{ se réduisant})\,,$$

$$\frac{(e' :_r n-1)_{e' | e \xrightarrow{r} e'}}{e : n}\ [\text{RED n}] \quad (e \text{ se réduisant}, n > 0)\,.$$

Tab. 2.8 – Sémantique par récriture : propriété finitaire stable

Remarquons que si la relation de réduction $\xrightarrow{r}$ n'est pas déterministe, dans la règle [RED n], le jugement $e :_r n$ se déduit de la conjonction des prémisses $(e' :_r n-1)_{e'|e\xrightarrow{r}e'}$. On peut imaginer une forme plus faible, où le jugement se déduirait de la disjonction de ces prémisses. Dans le cas qui nous intéresse, où la relation est déterministe, cela revient évidemment au même.
Une propriété finitaire stable est vérifiée universellement.

2.1.9 Proposition (Sémantique par récriture : Lemme des propriétés finitaires stables)

Soit $\xrightarrow{r}$ une relation de réduction totale. Considérons une propriété finitaire stable définie par un prédicat de jugements $(e :_r n)_{e,n}$. Alors elle est vérifiée universellement : pour tout programme e et tout entier n, on a

$$e :_r n\,.$$

Démonstration
Primo, on construit pour chaque programme e et chaque entier n une preuve dans le système d'inférence de la définition 2.1.8 (p. 128), secundo, on montre qu'il n'existe pas de preuves infinies dans ce système, ce qui fait que les preuves construites sont nécessairement finies, tertio, on conclut par la stabilité que la propriété est toujours vérifiée.

Définissons un système d'équations récursives, d'inconnues $(X_{e,n})_{e\in\Lambda^0,n\in\mathbb{N}}$, par les équations de la table 2.9. Comme la relation de réduction est totale,

$$X_{e,n} = \begin{cases} \dfrac{\emptyset}{e :_r n} & \text{si } e \text{ valeur,} \\ \dfrac{\emptyset}{e :_r n} & \text{si } e \text{ se réduit et } n = 0\,, \\ \dfrac{(X_{e',n-1})_{e'|e\xrightarrow{r}e'}}{e :_r n} & \text{si } e \text{ se réduit et } n > 0\,. \end{cases}$$

TAB. 2.9 – Dém. prop. 2.1.9

ce système est bien défini. Ce système est quasi-uniforme et compatible avec le système d'inférence de la définition 2.1.8. D'après la proposition 1.2.3 (p. 73), pour tout programme e et tout entier n, la valeur de l'unique solution en $X_{e,n}$ est une preuve suivant ce système d'inférence.

Montrons maintenant par récurrence sur l'entier n que toute preuve de conclusion $e :_r n$, e étant un programme quelconque, est finie.

- $n = 0$

Considérons une preuve de conclusion $e :_r 0$. Ce jugement est nécessairement la conclusion de la règle [VAL 0] ou de la règle [RED 0]. Dans les deux cas, il s'agit d'un axiome.

- $n > 0$

On suppose le résultat vrai en $n - 1$.
Considérons une preuve de conclusion $e :_r n$.
Si le jugement $e :_r n$ est la conclusion de la règle [VAL n], alors la preuve est finie, puisqu'il s'agit d'un axiome.
Sinon, le jugement $e :_r n$ a pour prémisses $(e' :_r n-1)_{e'|e \xrightarrow{r} e'}$, par la règle [RED n], et on applique l'hypothèse de récurrence à ces prémisses.

Finalement, pour tout programme e et tout entier n, le jugement $e :_r n$, qui admet une preuve finie, appartient à l'ensemble engendré inductivement par le système de la définition 2.1.8. Comme la propriété est stable, le jugement $e :_r n$ est vrai.

⌞

Ce lemme est le principal outil pour montrer des propriétés concernant les traces d'exécution des programmes. Ainsi, nous pouvons désormais montrer que les traces d'exécution sont maximales pour la relation de réduction.

2.1.10 Proposition (Traces d'exécution maximales pour $\xrightarrow{r}$)
Soit $\xrightarrow{r}$ une relation de réduction totale. Considérons un programme e et une trace t telle que $e \Downarrow_r t$. Alors la trace t correspond à une suite de réduction par $\xrightarrow{r}$ et est maximale pour $\xrightarrow{r}$: ou bien elle est infinie, ou bien elle se termine par une valeur.

Démonstration
Considérons la propriété finitaire suivante :

$$e :_r n \stackrel{def}{\Leftrightarrow} \forall t \in (\Lambda^0)^* \cup (\Lambda^0)^\omega \,.\, e \Downarrow_r t \wedge 0 < n < |t| \Rightarrow t(n-1) \xrightarrow{r} t(n)$$

On vérifie aisément que cette propriété est stable. Le seul cas intéressant concerne la règle [RED n].
Soit e un programme, n un entier strictement positif, et supposons que pour tout programme e' en lequel e se réduit, on ait $e' :_r n-1$. Montrons $e :_r n$.
Supposons $e \Downarrow_r t$ et $0 < n < |t|$. Par la règle [RED+] ou [RED-], il existe un programme e' et une trace t' tels que $e \xrightarrow{r} e'$, $e' \Downarrow_r t'$ et $t = e \rightarrow t'$. Comme on l'a remarqué, t' commence par e'.

Si $n = 1$, $t(n-1) = e$ et $t(n) = e'$, et on a bien $t(n-1) \xrightarrow{r} t(n)$.
Sinon, $t(n-1) = t'(n-2)$ et $t(n) = t'(n-1)$, et on applique l'hypothèse $e' :_r n-1$ pour conclure.
Finalement, la propriété est vérifiée par tout programme e et tout entier n.

Considérons un programme e et une trace t telle que $e \Downarrow_r t$. Si t est infinie, elle est maximale. Supposons qu'elle soit finie. Dans ce cas, la preuve de $e \Downarrow_r t$ est finie, et on peut facilement montrer par récurrence structurelle sur la preuve que la trace se termine par une valeur. Comme une valeur ne se réduit pas, par définition, la trace est bien maximale.
⌞

Nous supposons maintenant que la relation de réduction est totale et déterministe, comme la relation définie initialement. Il est alors facile de montrer qu'un programme possède une unique trace.

2.1.11 Proposition (Unicité de la trace)
Soit $\xrightarrow{r}$ une relation de réduction totale et déterministe. Considérons un programme e, et deux traces t_1 et t_2. Si le jugement $e \Downarrow_r t_1$ est valide, tout comme $e \Downarrow_r t_2$, alors les traces t_1 et t_2 sont égales.

Démonstration
Considérons la propriété finitaire suivante :

$$e :_r n \stackrel{def}{\Leftrightarrow} \forall t_1, t_2 \in (\Lambda^0)^* \cup (\Lambda^0)^\omega \,.\, e \Downarrow_r t_1 \wedge e \Downarrow_r t_2 \Rightarrow t_1(n) = t_2(n)$$

On vérifie aisément que cette propriété est stable. Le seul cas intéressant concerne la règle [RED n]. Comme la relation $\xrightarrow{r}$ est déterministe, elle n'est constituée que d'une prémisse. Supposons donc $e' :_r n-1$, avec $n > 0$, et $e \xrightarrow{r} e'$, et montrons $e :_r n$.
Soient t_1 et t_2 deux traces telles que $e \Downarrow_r t_1$ et $e \Downarrow_r t_2$. Par la règle [RED+] ou [RED-], il existe deux traces t'_1 et t'_2 telles que $e' \Downarrow_r t'_1$, $e' \Downarrow_r t'_2$, $t_1 = e \rightarrow t'_1$ et $t_2 = e \rightarrow t'_2$. De l'hypothèse $e' :_r n-1$, on déduit $t'_1(n-1) = t'_2(n-1)$, soit $t_1(n) = t_2(n)$.
Du lemme fondamental des propriétés finitaires stables, on déduit que la propriété est vérifiée par tout programme e et tout entier n, ce qui permet de conclure.
⌞

On notera $T_r(e)$ la trace d'exécution du programme e suivant le système d'inférence de la table 2.7 (p. 127), qui s'appuie sur la relation de réduction $\xrightarrow{r}$, définie dans la table 2.6 (p. 126).

2.1.3 Sémantique opérationnelle une

Il nous reste à montrer que les deux sémantiques présentées dans les tables 2.1 (p. 116) et 2.7 (p. 127) sont équivalentes, soit que pour tout programme e et toute trace t, on a

$$e \Downarrow t \Leftrightarrow e \Downarrow_r t\,.$$

Supposons que $e \Downarrow t_1$ et $e \Downarrow_r t_2$. On sait que t_2 commence par e. On va donc aussi le montrer pour t_1 : ce sera notre premier lemme. Ensuite, on sait que si e se réduit en e', alors t_2 commence par e, suivi de la trace de e' (pour le second système d'inférence). On va donc aussi le montrer pour t_1 : ce sera notre second lemme. On pourra conclure par récurrence sur la position dans la trace à l'égalité de t_1 et t_2.
On rappelle que pour tout programme e, on note $T(e)$ l'unique trace telle que $e \Downarrow T(e)$, et $T_r(e)$ l'unique trace telle que $e \Downarrow_r T_r(e)$.
Énonçons le premier lemme.

2.1.12 Lemme
Soient e un programme et t une trace tels que $e \Downarrow t$. Alors t commence par e.

Démonstration
Considérons la propriété finitaire suivante :

$$e : n \stackrel{def}{\Leftrightarrow} n = 0 \Rightarrow T(e)(n) = e\,.$$

Montrons qu'elle est stable pour le système d'inférence de la définition 2.1.4 (p. 121). Passons en revue les différentes règles d'inférence. Il s'agit à chaque fois de montrer que si les prémisses sont vérifiées, alors la conclusion aussi. Comme pour tout programme e et tout entier n, le jugement $e : n$ est trivialement vérifié dès que $n > 0$, on se limite au cas où n vaut zéro.
• [ABS 0]
C'est immédiat, puisque $\lambda x\, e \Downarrow \lambda x\, e$.
Considérons désormais deux programmes e_1 et e_2.
• [APP1 0]
Supposons $0 < |T(e_1)| - 1$ et $e_1 : 0$.
Comme dans ce cas $|T(e_1)| > 1$, on a :

$$\begin{aligned} T(e_1\, e_2)(0) &= T(e_1)(0)\, e_2 \\ &= e_1\, e_2 \quad (\text{hyp. } e_1 : 0)\,, \end{aligned}$$

et on peut conclure que $e_1\, e_2 \,:\, 0$ est vérifié.
• [APP1-2 0]
Supposons $0 = |T(e_1)| - 1$, $e_1 \,:\, 0$ et $e_2 \,:\, 0$.
Comme $|T(e_1)| = 1$ et $e_1 \,:\, 0$, on a $\overline{T(e_1)} = T(e_1)(0) = e_1$. Puis :

$$\begin{aligned} T(e_1\, e_2)(0) &= \overline{T(e_1)}\, T(e_2)(0) \\ &= e_1\, e_2 \quad (\text{hyp. } e_2 \,:\, 0)\,, \end{aligned}$$

ce qui permet de conclure que $e_1\, e_2 \,:\, 0$ est vérifié.
⌞

Voyons maintenant le second lemme.

2.1.13 Lemme
Supposons que e se réduise en e'. Soient t et t' deux traces telles que $e \Downarrow t$ et $e' \Downarrow t'$. Alors :

$$t = e \to t'\,.$$

Démonstration
Le programme e se décompose de manière unique en $E[(\lambda x\, a)\, v]$, qui se réduit en $E[a[v/x]]$. Le résultat se montre par récurrence sur E. On notera r le radical $(\lambda x\, a)\, v$, et r' son β-réduit.
• $E = -$
Le jugement $r \Downarrow T(r)$ est nécessairement la conclusion de la règle [APP+] ou [APP3-], avec pour prémisse droite $r' \Downarrow T(r')$, et donc $T(r) = r \to T(r')$.
• $E = E_1\, e_2$
Le jugement $e \Downarrow t$ est nécessairement la conclusion d'une des règles [APP+], [APP1-], [APP2-] ou [APP3-].
Considérons le cas de la règle [APP+] :

$$\frac{E_1[r] \Downarrow t_1 \quad e_2 \Downarrow t_2 \quad b[u/y] \Downarrow t_3}{E_1[r]\, e_2 \Downarrow <\mathrm{p}(t_1)\, e_2 > \to < \overline{t_1}\, t_2 > \to t_3}\,,$$

avec $\overline{t_1} = \lambda y\, b$ et $\overline{t_2} = u$. Par l'hypothèse de récurrence appliquée à $E_1[r]$, on a $t_1 = E_1[r] \to T(E_1[r'])$, d'où on déduit successivement

$$\begin{aligned} &E_1[r]\, e_2 \Downarrow E_1[r]\, e_2 \to <\mathrm{p}(T(E_1[r']))\, e_2 > \to < \overline{t_1}\, t_2 > \to t_3\,, \\ &E_1[r]\, e_2 \Downarrow E_1[r]\, e_2 \to T(E_1[r']\, e_2)\,, \end{aligned}$$

puisque $\overline{t_1} = \overline{T(E_1[r'])}$.
Les autres cas sont des variations de ce cas, et nous ne les traitons pas.
• $E = (\lambda y\, b)\, E_2$

Le jugement $e \Downarrow t$ est nécessairement la conclusion d'une des règles [APP+], [APP2-] ou [APP3-].
Considérons le cas de la règle [APP+] :

$$\frac{\lambda y\, b \Downarrow \lambda y\, b \quad E_2[r] \Downarrow t_2 \quad b[u/y] \Downarrow t_3}{(\lambda y\, b)\, E_2[r] \Downarrow (\lambda y\, b)\, t_2 \to t_3},$$

avec $\overline{t_2} = u$. Par l'hypothèse de récurrence appliquée à $E_2[r]$, on a $t_2 = E_2[r] \to T(E_2[r'])$, d'où on déduit successivement

$$(\lambda y\, b)\, E_2[r] \Downarrow (\lambda y\, b)\, E_2[r] \to\!< (\lambda y\, b)\, T(E_2[r']) >\!\to t_3\,,$$
$$(\lambda y\, b)\, E_2[r] \Downarrow (\lambda y\, b)\, E_2[r] \to T((\lambda y\, b)\, E_2[r'])\,,$$

puisque $\overline{t_2} = \overline{T(E_2[r'])}$.
Les autres cas sont des variations de ce cas, et nous ne les traitons pas.
⌞

Comme annoncé, on peut conclure à l'équivalence des deux sémantiques.

2.1.14 Théorème (Équivalence des sémantiques)
Pour tout programme e et toute trace t, on a :

$$e \Downarrow t \Leftrightarrow e \Downarrow_r t\,.$$

Démonstration
Considérons la propriété finitaire suivante :

$$e :_r n \quad \stackrel{def}{\Leftrightarrow} \quad T(e)(n) = T_r(e)(n)$$

Montrons qu'elle est stable (pour le système de la définition 2.1.8 (p. 128)). C'est immédiat, en utilisant le lemme 2.1.12 (p. 132) pour la règle [RED 0] et le lemme 2.1.13 (p. 133) pour la règle [RED n].
Les deux sémantiques sont bien équivalentes.
⌞

Ce qui intéresse avant tout dans l'exécution d'un programme, c'est le résultat, ou bien la valeur finale de la trace, si elle est finie, ou bien la divergence (ou la non-terminaison), si elle est infinie. Si, conformément à la tradition, on ajoute une valeur de non-terminaison, notée $\perp$, une trace t est donc abstraite en $\overline{t}$, suivant la notation déjà utilisée. Il est alors possible de récrire les deux systèmes d'évaluation précédents, en réalisant cette abstraction pour chaque règle.

$$\frac{}{\lambda x\, e \Downarrow \lambda x\, e}\ [\text{ABS}]$$

$$\frac{+\ e_1 \Downarrow v_1 \quad +\ e_2 \Downarrow v_2 \quad +\ e[v_2/x] \Downarrow v}{e_1\, e_2 \Downarrow v}\ [\text{APP+}] \quad \left(\begin{array}{l} v_1, v_2, v \in \mathbf{V} \\ v_1 = \lambda x\, e \end{array} \right)$$

$$\frac{-\ e_1 \Downarrow \bot}{e_1\, e_2 \Downarrow \bot}\ [\text{APP1-}]$$

$$\frac{+\ e_1 \Downarrow v_1 \quad -\ e_2 \Downarrow \bot}{e_1\, e_2 \Downarrow \bot}\ [\text{APP2-}] \quad (v_1 \in \mathbf{V})$$

$$\frac{+\ e_1 \Downarrow v_1 \quad +\ e_2 \Downarrow v_2 \quad -\ e[v_2/x] \Downarrow \bot}{e_1\, e_2 \Downarrow \bot}\ [\text{APP3-}] \quad \left(\begin{array}{l} v_1, v_2 \in \mathbf{V} \\ v_1 = \lambda x\, e \end{array} \right)$$

TAB. 2.10 – Sémantique par décomposition : résultats d'exécution

2.1.15 Définition et proposition (Résultat d'exécution)
Un résultat *est soit une valeur, soit la valeur de divergence, $\bot$.*
Un programme a pour résultat ρ si le jugement $e \Downarrow \rho$ est valide, dans le système de la table 2.10 (p. 135) ou celui de la table 2.11 (p. 136).
Le choix du système est indifférent et tout programme s'évalue en un et un seul résultat.

Démonstration
Comme tout programme possède une trace, il donne aussi au moins un résultat, obtenu par l'un des deux systèmes d'inférence précédent, égal à la valeur finale de la trace, ou à $\bot$. Pour chaque système, la seule question est l'unicité de ce résultat. Il est facile de montrer par récurrence structurelle sur les preuves finies que si un programme donne une valeur, alors il ne donne pas d'autre résultat. On en déduit l'unicité du résultat pour ces deux systèmes d'évaluation, puisqu'il n'y a qu'une valeur de divergence.
⌞

On pourra donc désigner par Eval(e) le résultat de l'évaluation du programme e, dans un des deux systèmes, indifféremment.

Pour conclure, il apparaît que la sémantique par récriture est beaucoup plus simple à manier que la sémantique par décomposition : moins de règles

$$\frac{\emptyset}{v \Downarrow_r v}\,[\text{VAL}] \quad (v \in \mathbf{V})$$

$$\frac{+\,e' \Downarrow_r v}{e \Downarrow_r v}\,[\text{RED+}] \quad \left(\begin{array}{l} e, e' \in \Lambda^0 \\ v \in \mathbf{V} \\ e \xrightarrow{\text{r}} e' \end{array}\right)$$

$$\frac{-\,e' \Downarrow_r \bot}{e \Downarrow_r \bot}\,[\text{RED-}] \quad \left(\begin{array}{l} e, e' \in \Lambda^0 \\ e \xrightarrow{\text{r}} e' \end{array}\right)$$

Tab. 2.11 – Sémantique par récriture : résultats d'exécution

d'inférence, méthode de raisonnement naturelle avec la récurrence sur la suite des réductions. Cependant, si l'on restreint la sémantique par décomposition aux programmes convergents, le système d'inférence se simplifie, et une méthode de raisonnement naturelle est disponible, la récurrence structurelle sur les preuves de l'évaluation.
Ces conclusions sont conformes à la pratique observée : la sémantique par récriture est souvent préférée pour sa simplicité, la sémantique par décomposition restreinte aux programmes convergents.
Cette restriction semble particulièrement nécessaire lorsque la sémantique présente un caractère non-déterministe, appelé à disparaître lors du raffinement menant à l'implémentation : par exemple, pour un langage purement fonctionnel, l'addition peut être considérée comme une opération commutative, et on peut préférer différer le choix de l'ordre d'évaluation. En cas de non-déterminisme, les traces d'exécution se composent par entrelacement, ce qui rend difficile l'expression d'une sémantique par décomposition définissant les traces d'exécution, et donc aussi la définition des propriétés finitaires stables. D'une part, l'abstraction, qui fait passer des traces aux résultats, est particulièrement utile dans ce cas, d'autre part, la difficulté de raisonner avec les programmes divergents incite à se limiter aux seuls programmes convergents.

2.2 L'observation comme dénotation

Par les systèmes d'inférence précédents, un résultat est attribué aux programmes, mais non aux termes possédant des variables libres. Il est cependant facile de passer des programmes aux termes, en envisageant les termes

dans tous les contextes d'utilisation possibles. Considérons un terme e faisant partie d'un programme. Lorsque le terme e doit être évalué[19], des valeurs ont été substituées aux variables libres de e. Il est donc suffisant de donner le résultat de l'évaluation de e après avoir remplacé ses variables libres par des valeurs quelconques. Nous appelons une substitution de valeurs un *environnement opérationnel*. Précisément, pour s'affranchir de la dépendance vis-à-vis des variables libres, on définit un environnement opérationnel comme une application de l'ensemble des variables dans l'ensemble des valeurs. Étant donné un environnement opérationnel γ, une variable x et une valeur v, on note $\gamma.(x : v)$ l'environnement obtenu en modifiant la liaison de x ainsi :

$$(\gamma.(x : v))(y) \stackrel{def}{=} \begin{cases} \gamma(y) & \text{si } y \neq x\,, \\ v & \text{sinon.} \end{cases}$$

Par la suite, on supposera que γ parcourt l'ensemble des environnements opérationnels, noté $\mathbf{V}^{\mathbf{X}}$. L'interprétation opérationnelle d'un terme e associe à tout environnement opérationnel γ le résultat de l'évaluation de $e[\gamma]$, noté $[\![e]\!]_\gamma$:

$$[\![e]\!]_\gamma \stackrel{def}{=} \mathrm{Eval}(e[\gamma])\,.$$

Cette interprétation peut être définie sous la forme d'une sémantique dénotationnelle. Munissons l'ensemble des résultats d'une application binaire map vérifiant, pour toutes valeurs v_1 et v_2 :

$$\begin{aligned} \mathrm{map}(\bot, \bot) &= \bot\,, \\ \mathrm{map}(\bot, v_2) &= \bot\,, \\ \mathrm{map}(v_1, \bot) &= \bot\,, \\ \mathrm{map}(v_1, v_2) &= \mathrm{Eval}(v_1\, v_2)\,. \end{aligned}$$

Autrement dit, l'application est stricte en chacun de ses deux arguments (ce qui, pour le second argument, correspond à un appel par valeur), et calcule le résultat de l'application du premier argument au second. Alors l'interprétation opérationnelle vérifie les équations suivantes, pour tout environnement opérationnel γ :

$$\begin{aligned} [\![x]\!]_\gamma &= \gamma(x)\,, \\ \mathrm{map}([\![\,\lambda x\, e]\!]_\gamma, v) &= [\![e]\!]_{\gamma.(x\,:\,v)} \quad (v \text{ valeur})\,, \\ [\![e_1\, e_2]\!]_\gamma &= \mathrm{map}([\![e_1]\!]_\gamma, [\![e_2]\!]_\gamma)\,. \end{aligned}$$

[19] Plus précisément, c'est un descendant du terme e qui est évalué.

Il ne s'agit pas véritablement d'une sémantique dénotationnelle, car la propriété d'extensionalité n'est pas vérifiée : si f et g sont deux valeurs, l'égalité pour toute valeur v de $\mathrm{map}(f, v)$ et $\mathrm{map}(g, v)$ n'implique pas que f et g soient égaux.
Par exemple, $\lambda x\, I$ et $\lambda x\, I\, I$ (I étant l'identité), appliqués à n'importe quelle valeur, donnent le même résultat, I, et cependant ce sont deux valeurs distinctes.
Aussi les équations précédentes ne permettent pas de construire par composition[20] l'interprétation opérationnelle d'un terme : la deuxième équation ne détermine pas de manière unique la dénotation de l'abstraction, $[\![\, \lambda x\, e]\!]_\gamma$.

L'interprétation obtenue par la sémantique opérationnelle donne des détails sur la structure interne, empêchant de développer une théorie riche de l'équivalence entre termes, deux termes étant équivalents s'ils ont la même interprétation :

$$e =_{\mathrm{op}} e' \quad \stackrel{def}{\Leftrightarrow} \quad \forall\, \gamma \in \mathbf{V}^{\mathbf{X}}.\ [\![e]\!]_\gamma = [\![e']\!]_\gamma\,.$$

Il est aussi possible de comparer les termes à partir de leur interprétation, une fois que l'ensemble des résultats est muni de la relation d'ordre définie par $\perp < v$, pour toute valeur v :

$$e \leq_{\mathrm{op}} e' \quad \stackrel{def}{\Leftrightarrow} \quad \forall\, \gamma \in \mathbf{V}^{\mathbf{X}}.\ [\![e]\!]_\gamma \leq [\![e']\!]_\gamma\,.$$

Voyons la validité de quelques règles classiques permettant de comparer opérationnellement des termes.
Le pré-ordre $\leq_{\mathrm{op}}$ n'est pas une congruence :

$$\frac{\emptyset}{x \leq_{\mathrm{op}} x} \qquad \text{est valide.}$$

$$\frac{e \leq_{\mathrm{op}} e'}{\lambda x\, e \leq_{\mathrm{op}} \lambda x\, e'} \qquad \text{n'est pas valide (prendre } e = I, e' = I\, I).$$

$$\frac{e_1 \leq_{\mathrm{op}} e_1' \quad e_2 \leq_{\mathrm{op}} e_2'}{e_1\, e_2 \leq_{\mathrm{op}} e_1'\, e_2'} \qquad \text{est valide.}$$

Elle vérifie la règle concernant la β-conversion, à condition de se limiter à des arguments déjà évalués (c'est la β_v-conversion) :

[20] La compositionalité est la possibilité de définir par récurrence structurelle sur les termes l'interprétation.

$$\frac{\emptyset}{(\lambda x\, e)\, v =_{\mathrm{op}} e[v/x]} \quad \text{est valide.}$$

Toute substitution par valeur est compatible avec le pré-ordre opérationnel :

$$\frac{e \leq_{\mathrm{op}} e'}{e[v/x] \leq_{\mathrm{op}} e'[v/x]} \quad \text{est valide.}$$

Enfin, l'η-conversion n'est pas valide :

$$\frac{\emptyset}{\lambda x\, e\, x = e} \quad \text{n'est pas valide (prendre } e \text{ divergent).}$$

Pour développer une théorie plus riche, il est préférable de s'appuyer sur l'observation des termes. Observer un terme, c'est l'utiliser pour observer les résultats qu'il donne, et non pas seulement l'évaluer.
Comme nous l'avons vu en introduction de ce chapitre, il existe une théorie maximale : c'est celle qui est induite par l'équivalence contextuelle. Nous allons définir l'observation de manière à induire la théorie maximale, tout en restreignant les contextes d'utilisation considérés : alors que l'équivalence contextuelle impose de placer chaque terme dans tout contexte d'utilisation possible, nous nous limitons aux contextes applicatifs, comme nous allons le voir.

2.2.1 Bisimilarité et équivalence contextuelle

Partons d'un terme e, dans un environnement opérationnel γ. On peut tout d'abord observer sa convergence, et s'il converge, observer son application à chaque valeur, et ainsi de suite. Cette observation peut être représentée par un arbre, dit *observationnel*. Les arbres observationnels sont les termes de la signature concrète suivante. Une seule sorte est utilisée, notée o ; l'ensemble des étiquettes est égal à la paire $\{\bot, \top\}$, $\bot$ représentant la divergence, $\top$ la convergence ; l'ensemble des positions est engendré par l'ensemble des valeurs, $\mathbf{V}$. Les étiquette $\bot$ et $\top$ ont le profil suivant :

$$\begin{aligned} \bot &: () \to o\,, \\ \top &: o^{\mathbf{V}} \to o\,. \end{aligned}$$

Voyons comment associer à un terme un arbre observationnel.

2.2.1 Définition et proposition (Observation d'un terme)
Il existe une unique application associant à tout terme e et tout environnement opérationnel γ un arbre observationnel, noté $\mathrm{Obs}(e)(\gamma)$*, et vérifiant :*

(i) si $e[\gamma]$ diverge, alors $\mathrm{Obs}(e)(\gamma) = \bot$,
(ii) si $e[\gamma]$ converge, alors $\mathrm{Obs}(e)(\gamma) = \top\ (\mathrm{Obs}(e\,v)(\gamma))_{v\in \boldsymbol{V}}$.

L'arbre observationnel $\mathrm{Obs}(e)(\gamma)$ *est appelé l'*observation *de e dans l'environnement opérationnel γ. L'application qui à γ associe* $\mathrm{Obs}(e)(\gamma)$*, notée* $\mathrm{Obs}(e)$*, est appelée l'observation de e.*
L'observation d'un programme e est indépendante de l'environnement d'observation et on note $\mathrm{Obs}^0(e)$ *la valeur qu'elle prend en tout environnement.*

Montrons dans un premier temps l'existence et l'unicité de l'application définissant l'observation.

Démonstration
Définissons un système d'équations récursives d'inconnues $(X_{(e,\gamma)})_{(e,\gamma)\in\Lambda^0\times\mathbf{V}^{\mathbf{X}}}$ par les équations suivantes :

$$X_{(e,\gamma)} = \begin{cases} \bot & \text{si } e[\gamma] \text{ diverge,} \\ \top\ (X_{(e\,v,\gamma)})_{v\in\mathbf{V}} & \text{si } e[\gamma] \text{ converge.} \end{cases}$$

Si pour tout terme e et tout environnement γ, $\mathrm{Obs}(e)(\gamma)$ vérifie les conditions (i) et (ii), alors la valuation ν définie par $\nu(X_{(e,\gamma)}) = \mathrm{Obs}(e)(\gamma)$ pour tout couple (e,γ) est solution du système. Comme le système admet une unique solution, il existe une unique application vérifiant (i) et (ii).
⌞

Avant de justifier que l'observation d'un programme ne dépend pas de l'environnement, il est utile de définir un mode de raisonnement co-inductif pour les observations, permettant de montrer l'égalité de deux observations. Il définit un ensemble de conditions suffisantes pour construire des preuves de l'égalité des arbres observationnels, égalité définie co-inductivement par le système suivant :

$$\frac{\emptyset}{(\bot,\bot)},$$

$$\frac{(a_{/v}, b_{/v})_{v\in\mathbf{V}}}{(a,b)} \quad (a,b \text{ arbres observationnels } \neq \bot).$$

2.2.2 Lemme (Raisonnement co-inductif pour les observations)
Soit R une relation définie sur les couples formés d'un terme et d'un environnement opérationnel, et vérifiant pour tout couple $((e,\gamma),(e',\gamma'))$ appartenant à R :

(i) $e[\gamma]$ converge si et seulement si $e'[\gamma']$ converge,

(ii) si $e[\gamma]$ converge, alors pour toute valeur v, $((e\,v,\gamma),(e'\,v,\gamma'))$ appartient à R.

Alors, pour tous termes e et e', et tous environnements γ et γ' tels que $((e,\gamma),(e',\gamma'))$ appartient à R,

$$\mathrm{Obs}(e)(\gamma) \quad = \quad \mathrm{Obs}(e')(\gamma')\,.$$

Démonstration
Pour chaque couple $((e,\gamma),(e',\gamma'))$ de R, on construit une preuve de l'égalité $\mathrm{Obs}(e)(\gamma) = \mathrm{Obs}(e')(\gamma')$, en résolvant le système d'équations récursives d'inconnues $(X_c)_{c\in R}$ et formé des équations suivantes :

$$X_{(e,\gamma),(e',\gamma')} = \begin{cases} \dfrac{\emptyset}{(\mathrm{Obs}(e)(\gamma),\mathrm{Obs}(e')(\gamma'))} & \text{si } e[\gamma] \text{ diverge,} \\[2ex] \dfrac{(X_{(e\,v,\gamma),(e'\,v,\gamma')})_{v\in\mathbf{V}}}{(\mathrm{Obs}(e)(\gamma),\mathrm{Obs}(e')(\gamma'))} & \text{si } e[\gamma] \text{ converge.} \end{cases}$$

Ce système est bien défini grâce à la seconde hypothèse sur R.
Pour la première équation, on sait par la première hypothèse sur R, que $e'[\gamma']$ diverge aussi, et donc $\mathrm{Obs}(e)(\gamma) = \mathrm{Obs}(e')(\gamma') = \bot$. Pour la seconde équation, on sait que $e'[\gamma']$ converge aussi, et donc $\mathrm{Obs}(e)(\gamma)$ et $\mathrm{Obs}(e')(\gamma')$ ne sont pas des observations triviales.
Ce système, quasi-uniforme et compatible avec le système d'inférence définissant l'égalité, définit donc des preuves de l'égalité des observations, par la proposition 1.2.3 (p. 73).
⌞

Terminons la démonstration de la proposition incluse dans la définition des observations.

Démonstration
On doit montrer que l'observation d'un programme ne dépend pas de l'environnement d'observation.
Soit R la relation définie par :

$$(e,\gamma)\ R\ (e',\gamma') \quad \stackrel{def}{\Leftrightarrow} \quad e = e' \wedge e \in \Lambda^0\,.$$

Les hypothèses du lemme 2.2.2 (p. 141) sont trivialement vérifiées. Ainsi, étant donné deux environnements, γ et γ', pour tout programme e, on a $\mathrm{Obs}(e)(\gamma) = \mathrm{Obs}(e)(\gamma')$.

⌞

La définition des observations aurait pu se limiter aux programmes, puisqu'on peut fermer n'importe quel terme par un environnement opérationnel pour l'observer.

2.2.3 Proposition (Fermeture d'un terme pour l'observation)
Soient e un terme et γ un environnement opérationnel. Observer e dans l'environnement γ est équivalent à observer le programme $e[\gamma]$:

$$\mathrm{Obs}(e)(\gamma) \quad = \quad \mathrm{Obs}^0(e[\gamma])\,.$$

Démonstration
Soit R la relation définie par :

$$(e,\gamma)\ R\ (e',\gamma') \quad \stackrel{def}{\Leftrightarrow} \quad e[\gamma] = e'\,.$$

Les hypothèses du lemme 2.2.2 (p. 141) sont trivialement vérifiées. On a donc pour tout programme e et tous environnements opérationnels γ et γ' :

$$\begin{array}{rcl} \mathrm{Obs}(e)(\gamma) & = & \mathrm{Obs}(e[\gamma])(\gamma') \\ & = & \mathrm{Obs}^0(e[\gamma]) \quad (e[\gamma] \in \Lambda^0)\,. \end{array}$$

⌞

L'observation est compatible avec l'équivalence opérationnelle.

2.2.4 Proposition (Compatibilité de l'observation avec l'équivalence opérationnelle)
Considérons deux termes e et e' équivalents opérationnellement. Alors ils mènent aux mêmes observations :

$$\frac{e =_{\mathrm{op}} e'}{\mathrm{Obs}(e) = \mathrm{Obs}(e')}\,.$$

Démonstration
Soit R la relation définie par :

$$(e,\gamma)\ R\ (e',\gamma') \quad \stackrel{def}{\Leftrightarrow} \quad \gamma = \gamma' \wedge e =_{\mathrm{op}} e'\,.$$

Les hypothèses du lemme 2.2.2 (p. 141) sont trivialement vérifiées. On en déduit l'implication demandée.
⌞

À l'aide des observations, on peut construire une première notion d'équivalence entre termes. Deux termes sont équivalents s'ils mènent aux mêmes observations :

$$e =_{\mathrm{B}} e' \quad \overset{def}{\Leftrightarrow} \quad \mathrm{Obs}(e) = \mathrm{Obs}(e')\,.$$

On dit que les deux termes sont *bisimilaires* (applicativement ou observationnellement).
Plutôt que seulement considérer cette équivalence entre les termes, il est préférable de définir un pré-ordre, la relation de *similarité*, dont la symétrisation donne l'équivalence. Ce pré-ordre s'appuie sur une relation d'ordre entre arbres d'observations, par laquelle on mesure le degré de convergence. Cette relation d'ordre est engendrée co-inductivement par le système d'inférence suivant, qui rappelle celui de l'égalité entre arbres observationnels :

$$\frac{\emptyset}{(\bot, a)} \quad (a \text{ arbre observationnel})\,,$$

$$\frac{(a_{/v}, b_{/v})_{v\in\mathbf{V}}}{(a,b)} \quad (a, b \neq \bot)\,.$$

On vérifie aisément que la symétrisation de cette relation donne l'égalité précédemment définie. Aussi, comme nous l'avons vu au premier chapitre (cf. §1.2.2 (p. 79)), cette relation peut se définir comme extension aux arbres (en tant que fonctions) de la relation d'ordre définie par $\bot < \top$:

$$a \leq b \Leftrightarrow \forall\, V \in \mathbf{V}^*\,.\ a(V) \leq b(V)\,.$$

La relation de similarité se définit alors ainsi :

$$e \leq_{\mathrm{B}} e' \quad \overset{def}{\Leftrightarrow} \quad \mathrm{Obs}(e) \leq \mathrm{Obs}(e')\,,$$

où la relation d'ordre entre les observations se déduit naturellement de la relation d'ordre définie pour les arbres d'observation :

$$\mathrm{Obs}(e) \leq \mathrm{Obs}(e') \quad \overset{def}{\Leftrightarrow} \quad \forall\, \gamma \in \mathbf{V}^{\mathbf{X}}\,.\ \mathrm{Obs}(e)(\gamma) \leq \mathrm{Obs}(e')(\gamma)\,.$$

Lorsque $e \leq_{\mathrm{B}} e'$, on dit que e' *simule* e.
L'ensemble des arbres observationnels possède un plus petit élément, $\bot$, et un plus grand, solution de l'équation suivante :

$$x = \top(x)_{v \in \mathbf{V}}\,.$$

Ces deux arbres observationnels correspondent à des observations de programmes, comme l'indique la proposition[21] suivante.

2.2.5 Proposition (Observations minimale et maximale)
Il existe un programme, noté $\mathrm{p_{max}}$*, vérifiant :*

$$\mathrm{Obs}^0(\mathrm{p_{max}}) = \top(\mathrm{Obs}^0(\mathrm{p_{max}}))_{v \in \boldsymbol{V}}\,.$$

Il existe un programme, noté $\mathrm{p_{min}}$*, tel que :*

$$\mathrm{Obs}^0(\mathrm{p_{min}}) = \bot\,.$$

L'idée est de prendre pour $\mathrm{p_{min}}$ un programme divergent et pour $\mathrm{p_{max}}$ le programme infini $\lambda y\,\lambda y\ldots$, qu'on définit comme point fixe du combinateur K, égal à $\lambda x\,\lambda y\,x$.

Démonstration
• Définition de $\mathrm{p_{max}}$
Soit $K = \lambda x\,\lambda y\,x$. Définissons un combinateur de point fixe :

$$Y \stackrel{def}{=} \lambda f\,B(f)\,,$$

où pour tout terme f,

$$B(f) \stackrel{def}{=} C(f)\,C(f)\,,$$

$C(f)$ étant défini par

$$C(f) \stackrel{def}{=} \lambda x\,\eta(f\,(x\,x)) \quad (x \notin \mathbf{FV}(f))\,,$$

et $\eta(f)$ réalisant l'η-expansion de f, soit

$$\eta(f) \stackrel{def}{=} \lambda x\,f\,x \quad (x \notin \mathbf{FV}(f))\,.$$

Définissons ainsi $\mathrm{p_{max}}$:

$$\mathrm{p_{max}} \stackrel{def}{=} \eta(K\,B(K))\,.$$

[21] C'est l'adaptation à l'appel par valeur d'une proposition de l'article d'Abramsky développant la théorie du λ-calcul paresseux (cf. [4, Prop. 2.7]).

On vérifie facilement que pour toute valeur v, $\mathrm{p}_{\max}\, v$ se réduit en $\mathrm{p}_{\max}$. On en déduit le résultat.

• Définition de $\mathrm{p}_{\min}$

Il suffit de définir $\mathrm{p}_{\min}$ ainsi :

$$\mathrm{p}_{\min} \stackrel{def}{=} (\lambda x\, x\, x)\, (\lambda x\, x\, x)\,.$$

Ainsi $\mathrm{p}_{\min}$ se réduit en lui-même et diverge.

⌞

Idéalement, la bisimilarité devrait correspondre à l'équivalence contextuelle, qui égale deux termes lorsqu'aucun contexte d'utilisation ne permet de les distinguer. De même, la similarité devrait correspondre au pré-ordre pour lequel un terme est inférieur à un autre lorsqu'aucun contexte d'utilisation ne permet d'observer la convergence du premier et la divergence du second. C'est ce que nous allons voir maintenant.

Étant donné une expression C, le *contexte* à une place associé à C[22] est l'application qui à tout terme e associe le terme obtenu par la greffe[23] dans C de e en toute occurrence de la place $-$[24]. On note $C[e]$ le résultat de la greffe (qui pourra en certaines occasions être considéré comme une expression, et non un terme), et $C_{[\ ]}$ le contexte.

Par exemple, si $C_1 = \lambda x -$, $C_2 = \lambda y -$ et $e = x$, alors $C_1[e]$ est l'identité, $\lambda x\, x$, et $C_2[e]$ le terme $\lambda y\, x$[25].

Le fait qu'on ne puisse pas distinguer deux termes dans un contexte donné peut s'exprimer ainsi : les deux programmes obtenus en plaçant les deux termes dans le contexte ou bien divergent ensemble, ou bien convergent ensemble. Autrement dit, seule la convergence peut être observée. Se précise ainsi la notion d'*équivalence contextuelle* présentée en introduction du chapitre.

2.2.6 Définition (Équivalence contextuelle)

Deux termes e et e' sont équivalents contextuellement *si dans tout contexte $C_{[\ ]}$ faisant de $C[e]$ et $C[e']$ des programmes, $C[e]$ converge si et seulement*

[22] L'application associant le contexte à l'expression ne passe pas au quotient par l'α-conversion, car un contexte (en tant qu'expression) peut lier les variables libres des termes greffés.

[23] La greffe, à la différence de la substitution, autorise la capture des variables libres par le contexte, lors du remplacement de la place par le terme.

[24] On rappelle que cette variable est réservée à ce seul usage, ce qui signifie qu'elle ne peut pas être liée.

[25] On remarque que C_1 et C_2 représentent le même terme, mais $C_{1[\ ]}$ et $C_{2[\ ]}$ sont deux contextes différents, puisque $C_1[e] \neq C_2[e]$.

si $C[e']$ converge.

$$e =_{\mathrm{C}} e' \stackrel{def}{\Leftrightarrow} \forall C_{[\ ]}\,. \\ C[e], C[e'] \in \Lambda^0 \Rightarrow (C[e] \Downarrow \perp \Leftrightarrow C[e'] \Downarrow \perp)\,.$$

Associé à cette équivalence, un *pré-ordre contextuel* peut être défini.

2.2.7 Définition (Pré-ordre contextuel)
Un terme e converge contextuellement moins (ou diverge contextuellement plus) qu'un terme e' si pour tout contexte $C_{[\ ]}$ faisant de $C[e]$ et $C[e']$ des programmes, la convergence de $C[e]$ implique celle de $C[e']$.

$$e \leq_{\mathrm{C}} e' \stackrel{def}{\Leftrightarrow} \forall C_{[\ ]}\,. \\ C[e], C[e'] \in \Lambda^0 \Rightarrow (C[e'] \Downarrow \perp \Rightarrow C[e] \Downarrow \perp)\,.$$

L'équivalence contextuelle est donc la symétrisation du pré-ordre contextuel :

$$=_{\mathrm{C}} \;=\; \leq_{\mathrm{C}} \cap\, {}^{t}\!\leq_{\mathrm{C}}\ .$$

Intuitivement, il est clair que l'équivalence contextuelle entraîne l'égalité des observations, puisque moins de contextes sont en jeu lorsque se construit l'observation. Précisément, supposons que e et e' soient équivalents contextuellement. Pour obtenir l'égalité des observations de e et e', on va raisonner co-inductivement, comme précédemment. On aimerait donc avoir, pour tout environnement opérationnel γ :

- $e[\gamma]$ converge si et seulement si $e'[\gamma]$ converge,
- si $e[\gamma]$ converge, alors pour toute valeur v, $e[\gamma]\,v$ et $e'[\gamma']\,v$ sont contextuellement équivalents,

si bien que le lemme 2.2.2 (p. 141) pourrait alors s'appliquer.
Établissons donc ces deux propriétés de l'équivalence contextuelle, en les généralisant et en les formulant à partir du pré-ordre contextuel.
Définissons tout d'abord une congruence.

2.2.8 Définition ((Pré-)Congruence)
Une relation définie sur l'ensemble des termes est une pré-congruence *si elle est stable pour le système d'inférence composé de toutes les règles des formes*

suivantes :

$$\frac{\emptyset}{(x,x)}\,[\text{CONG-VAR}] \quad (x \text{ var.})\,,$$

$$\frac{(e,e')}{(\lambda x\, e, \lambda x\, e')}\,[\text{CONG-ABS}]\,,$$

$$\frac{(e_1,e_1') \quad (e_2,e_2')}{(e_1\, e_2, e_1'\, e_2')}\,[\text{CONG-APP}]\,.$$

C'est une congruence *si c'est de plus une relation d'équivalence.*

Une autre formulation de la propriété de congruence utilise des contextes à plusieurs places, les places étant des variables réservées à ce seul usage, et notées $-_1, \ldots, -_n, \ldots$. Si C est une expression, le *contexte* à n places associé, noté $C_{[n]}$, est l'application qui à tout n-uplet de termes $(e_1, \ldots, e_n)$ associe le terme obtenu par la greffe simultanée dans C de e_1 en chaque occurrence de $-_1$, de e_2 en chaque occurrence de $-_2$, etc. Cette image est notée $C[e_1, \ldots, e_n]$. Bien noter que n peut prendre la valeur nulle, définissant alors un contexte à zéro place, l'image $C[]$ étant alors égale à C.

2.2.9 Proposition (Pré-congruence et passage au contexte)
Une relation définie sur l'ensemble des termes est une pré-congruence si et seulement si elle est stable pour le système d'inférence composé de toutes les règles de la forme suivante :

$$\frac{(e_1,e_1') \quad \ldots \quad (e_n,e_n')}{(C[e_1,\ldots,e_n], C[e_1',\ldots,e_n'])}\,[\text{CONG-CONT}] \quad \left(\begin{array}{l} n \geq 0 \\ C_{[n]} \text{ contexte.} \end{array}\right).$$

Démonstration
La stabilité dans ce système d'inférence est évidemment suffisante. Pour montrer qu'elle est nécessaire, on procède par récurrence structurelle sur le contexte, sans aucune difficulté.
⌞

Il est clair que l'équivalence contextuelle est une congruence.

2.2.10 Proposition (Congruence de l'équivalence contextuelle)
Le pré-ordre contextuel est une pré-congruence et l'équivalence contextuelle une congruence.

Démonstration
Examinons les différentes règles.
• [CONG-VAR]
C'est immédiat.
• [CONG-ABS]
Immédiat.
• [CONG-APP]
Supposons $e_1 \leq_{\mathrm{C}} e_1'$ et $e_2 \leq_{\mathrm{C}} e_2'$. On doit montrer que $e_1\, e_2 \leq_{\mathrm{C}} e_1'\, e_2'$.
Soit $C_{[\ ]}$ un contexte faisant de $e_1\, e_2$ et de $e_1'\, e_2'$ des programmes. On a successivement :

$$C[e_1'\, e_2'] \Downarrow \perp \Rightarrow C[e_1\, e_2'] \Downarrow \perp \Rightarrow C[e_1\, e_2] \Downarrow \perp,$$

en appliquant l'hypothèse aux contextes $(C[-\, e_2'])_{[\ ]}$ et $(C[e_1\, -])_{[\ ]}$[26].
Le pré-ordre contextuel est donc une pré-congruence. Par symétrie, l'équivalence contextuelle est aussi une pré-congruence ; comme c'est une relation d'équivalence, c'est bien une congruence.
⌞

Voyons maintenant le passage au quotient des substitutions par valeurs.

2.2.11 Proposition (Équivalence contextuelle : passage au quotient des substitutions par valeurs)

Soient e et e' deux termes, et v et v' deux valeurs.

- *Supposons que e et v convergent contextuellement moins que e' et v' respectivement. Alors $e[v/x]$ converge contextuellement moins que $e'[v'/x]$.*

$$\frac{e \leq_{\mathrm{C}} e' \quad v \leq_{\mathrm{C}} v'}{e[v/x] \leq_{\mathrm{C}} e'[v'/x]} \quad (v, v' \in \boldsymbol{V}).$$

- *Supposons que e et e' soient équivalents contextuellement, de même que v et v'. Alors $e[v/x]$ et $e'[v'/x]$ sont équivalents contextuellement.*

$$\frac{e =_{\mathrm{C}} e' \quad v =_{\mathrm{C}} v'}{e[v/x] =_{\mathrm{C}} e'[v'/x]} \quad (v, v' \in \boldsymbol{V}).$$

Démonstration
Supposons $e \leq_{\mathrm{C}} e'$ et $v \leq_{\mathrm{C}} v'$. Considérons un contexte $C_{[\ ]}$ faisant de $C[e[v/x]]$ et de $C[e'[v'/x]]$ des programmes, ce qui implique que C ne possède aucune variable libre (autre que la place $-$). On veut montrer que

[26] $C[-\, e_2']$ et $C[e_1\, -]$ représentent ici les expressions obtenues par greffe, et non les termes.

si $C[e'[v'/x]]$ diverge, alors $C[e[v/x]]$ diverge. Comme les variables libres de $e[v/x]$ et $e'[v'/x]$ sont distinctes de x, les programmes $C[e[v/x]]$ et $C[e'[v'/x]]$ sont α-convertibles en deux programmes, $C'[e[v/x]]$ et $C'[e'[v'/x]]$, où C' est une expression dont les variables liées sont distinctes de x.
On remarque que $(\lambda x\, C'[e])\, v$ se réduit en $C'[e[v/x]]$, comme $(\lambda x\, C'[e'])\, v'$ en $C'[e'[v'/x]]$. Il est donc équivalent de montrer que si $(\lambda x\, C'[e'])\, v'$ diverge, alors $(\lambda x\, C'[e])\, v$ diverge.
Cette implication découle de l'inégalité $(\lambda x\, C'[e])\, v \leq_{\mathrm{C}} (\lambda x\, C'[e'])\, v'$, conclusion valide de la règle [CONG-CONT], avec le contexte $((\lambda x\, C'[-_1])\, -_2)_{[2]}$ puisque $e \leq_{\mathrm{C}} e'$ et $v \leq_{\mathrm{C}} v'$.
La propriété pour l'équivalence contextuelle se déduit par symétrie.
⌞

Pour l'observation des termes, et le système d'équations récursives la définissant, plutôt que raisonner par unicité de la solution, on préfère raisonner par co-induction directement à partir de relations, suivant un principe voisin du lemme 2.2.2 (p. 141) déjà vu pour l'égalité.

2.2.12 Proposition (Raisonnement co-inductif pour les simulations)
Soit R une simulation*, c'est-à-dire une relation entre termes vérifiant :*

(i) pour tout couple (e, e') de R, et tout environnement γ, si $e[\gamma]$ converge, alors $e'[\gamma]$ converge,

(ii) pour tout couple (e, e') de R, et pour toute valeur v, le couple $(e\, v, e'\, v)$ appartient à R.

Alors la relation R est incluse dans la relation de similarité :

$$\forall\, e, e'\,.\; e\; R\; e' \Rightarrow e \leq_{\mathrm{B}} e'\,.$$

Démonstration
Pour tout couple (e, e') de R et tout environnement γ, on construit une preuve de $\mathrm{Obs}(e)(\gamma) \leq \mathrm{Obs}(e')(\gamma)$, en définissant un système d'équations récursives, d'inconnues $(X_{(e,e'),\gamma})_{(e,e')\in R,\gamma\in\mathbf{V}^{\mathbf{X}}}$ et d'équations :

$$X_{(e,e'),\gamma} = \begin{cases} \dfrac{\emptyset}{(\mathrm{Obs}(e)(\gamma), \mathrm{Obs}(e')(\gamma))} & \text{si } \mathrm{Obs}(e)(\gamma) = \bot\,, \\[2ex] \dfrac{(X_{(e\,v,e'\,v),\gamma})_{v\in\mathbf{V}}}{(\mathrm{Obs}(e)(\gamma), \mathrm{Obs}(e')(\gamma))} & \text{si } \mathrm{Obs}(e)(\gamma) \neq \bot\,. \end{cases}$$

Par la seconde hypothèse sur R, le système est bien défini. Pour la seconde équation, si $\mathrm{Obs}(e)(\gamma) \neq \bot$, alors par la première hypothèse, $\mathrm{Obs}(e')(\gamma) \neq \bot$.

Ce système quasi-uniforme est donc compatible avec le système définissant l'ordre entre les arbres d'observations. D'après la proposition 1.2.3 (p. 73), il en résulte que pour tout couple (e, e') de R et tout environnement γ, la valeur de l'unique solution en $((e, e'), \gamma)$ est une preuve de $\mathrm{Obs}(e)(\gamma) \leq \mathrm{Obs}(e')(\gamma)$.
⌞

Comme annoncé, on montre maintenant que l'équivalence contextuelle entraîne la bisimilarité.

2.2.13 Proposition (Équivalence contextuelle implique bisimilarité)
Soient e_1 et e_2 deux termes. Si e_1 converge contextuellement moins que e_2, alors e_2 simule e_1 :

$$\forall\, e_1, e_2\,.\ e_1 \leq_{\mathrm{C}} e_2 \Rightarrow e_1 \leq_{\mathrm{B}} e_2\,.$$

Si e_1 et e_2 sont équivalents contextuellement, alors ils sont bisimilaires :

$$\forall\, e_1, e_2\,.\ e_1 =_{\mathrm{C}} e_2 \Rightarrow e_1 =_{\mathrm{B}} e_2\,.$$

Démonstration
On applique la proposition 2.2.12 (p. 149). Les conditions sont vérifiées par le pré-ordre contextuel, puisque ce sont des conséquences des propositions 2.2.11 (p. 148) et 2.2.10 (p. 147) respectivement. Par symétrie, on déduit la propriété pour l'équivalence contextuelle.
⌞

La réciproque est plus délicate, comme on pouvait s'y attendre. C'est l'objet du paragraphe suivant que de la montrer.

2.2.2 La méthode de Howe

Suivant la méthode de Howe, on va montrer que la similarité est une pré-congruence. Dans ce cas, si e' simule e, pour tout contexte $C_{[\,]}$, le programme $C[e']$ simule $C[e]$, et donc $C[e]$ diverge si $C[e']$ diverge. Puis, par symétrie, la bisimilarité égale l'équivalence contextuelle.
Pour montrer la propriété de pré-congruence, on étend la relation de similarité de manière à obtenir une pré-congruence et à vérifier les conditions de la proposition 2.2.12 (p. 149). Par co-induction, on déduit que cette extension est contenue dans la relation de similarité : elle est donc égale, ce qui fait de la similarité une pré-congruence.
La difficulté réside dans la définition de l'extension. La similarité vérifie évidemment les conditions de la proposition 2.2.12 (p. 149). Des propriétés de

congruence, seule la règle concernant l'application pose problème : si e'_1 et e'_2 simulent respectivement e_1 et e_2, il n'est pas évident que $e'_1\, e'_2$ simule $e_1\, e_2$. Récapitulons les propriétés que doit vérifier une extension potentielle R :

- [EXT] : $\leq_{\mathrm{B}} \subseteq R$,
- [CONG] : R est une pré-congruence,
- si le couple de termes (e, e') appartient à R,
  - [CO-IND1] : pour tout environnement γ, si $e[\gamma]$ converge, alors $e'[\gamma]$ converge,
  - [CO-IND2] : pour toute valeur v, $(e\, v, e'\, v)$ appartient à R.

La dernière condition [CO-IND2] se déduit des propriétés de pré-congruence, et est donc inutile.

Considérons un couple de termes (e, e') de R et un environnement γ. On peut penser que $(e[\gamma], e'[\gamma])$ appartient encore à R : c'est une nouvelle condition, [ENV]. Supposons que $e[\gamma]$ converge vers $\lambda x\, a$. Il est raisonnable de supposer que le couple $(\lambda x\, a, e'[\gamma])$ appartienne aussi à R : c'est une dernière condition, notée [RED]. On aimerait pouvoir déduire de $\lambda x\, a\ R\ e'[\gamma]$ la convergence de $e'[\gamma]$, ce qui donnerait [CO-IND1]. Tentons de définir R par un ensemble de règles d'inférence. Essayons donc, assez naturellement, la règle

$$\frac{(\lambda x\, a, \lambda x\, a')}{(\lambda x\, a, e'[\gamma])} \quad (e'[\gamma] \xrightarrow{\mathrm{r}}{}^{*}\ \lambda x\, a')\,,$$

ou, mieux, en combinaison avec la règle de congruence concernant l'abstraction, de manière à rendre le système déterministe (un jugement (e, e') ne pouvant alors être la conclusion que d'une seule règle), assurant ainsi que le jugement $(\lambda x\, a, e'[\gamma])$ se déduit bien de cette règle,

$$\frac{(a, a')}{(\lambda x\, a, e'[\gamma])} \quad (e'[\gamma] \xrightarrow{\mathrm{r}}{}^{*}\ \lambda x\, a')\,.$$

On peut généraliser cette formulation aux deux autres règles de congruence. Ainsi, la relation R définie inductivement par ces règles d'inférence vérifie [CONG] et [CO-IND1]. Il reste à vérifier [EXT], [ENV] et [RED]. Pour définir une extension, le plus simple est de remplacer la fermeture réflexive et transitive de la relation de réduction par la similarité, qui la contient. Il s'avère que les conditions [ENV] et [RED] sont alors vérifiées, comme nous allons le voir formellement.

L'extension met en relation un terme avec tous les termes de même structure, modulo la similarité. Cette construction se généralise à tout préordre.

$$\frac{\emptyset}{(x,e)} \quad (x \preceq e)$$

$$\frac{(b,b')}{(\lambda x\, b, e)} \quad (\lambda x\, b' \preceq e)$$

$$\frac{(a_1,e_1) \quad (a_2,e_2)}{(a_1\, a_2, e)} \quad (e_1\, e_2 \preceq e)$$

TAB. 2.12 – Extension de Howe

2.2.14 Définition (Extension de Howe d'un pré-ordre)
Soit $\preceq$ une relation de pré-ordre (réflexive et transitive). L'extension de Howe du pré-ordre $\preceq$, notée $\preceq^\bullet$ est la relation binaire sur les termes engendrée inductivement par le système d'inférence de la table 2.12 (p. 152).

Établissons quelques propriétés universelles de $\preceq^\bullet$.

2.2.15 Proposition (Propriétés universelles de l'extension de Howe)
L'extension de Howe $\preceq^\bullet$ du pré-ordre $\preceq$ vérifie les propriétés suivantes :
- *La relation $\preceq^\bullet$ est réflexive.*
- *La relation $\preceq^\bullet$ est stable par composition à droite avec le pré-ordre $\preceq$:*
$$\frac{e \preceq^\bullet e'}{e \preceq^\bullet e''} \quad (e' \preceq e'')\,.$$
- *La relation $\preceq^\bullet$ est une extension du pré-ordre $\preceq$:*
$$\preceq \subseteq \preceq^\bullet .$$
- *La relation $\preceq^\bullet$ est une pré-congruence.*

Démonstration
• Réflexivité
Il est facile de construire une preuve de $e \preceq^\bullet e$, par récurrence structurelle sur e, en utilisant la réflexivité de $\preceq$.
• $(\preceq^\bullet \circ \preceq) \subseteq \preceq^\bullet$
C'est immédiat, par récurrence structurelle sur la preuve de $e \preceq^\bullet e'$, en utilisant la transitivité de $\preceq$.
• $\preceq \subseteq \preceq^\bullet$
On a $e \preceq^\bullet e$, par réflexivité, et si $e \preceq e'$, alors d'après la propriété précédente,

$e \preceq^\bullet e'$.
• Pré-congruence
Il suffit de considérer la définition, et d'utiliser la réflexivité de $\preceq$.
⌞

Désormais, nous nous intéressons à l'extension de Howe associée à la similarité, $\leq_{\mathrm{B}}{}^\bullet$.
Avant d'étudier le comportement de cette extension relativement à la réduction, il est utile de connaître celui de $\leq_{\mathrm{B}}$.
Tout d'abord, intéressons-nous aux substitutions.

2.2.16 Lemme (Similarité : compatibilité des substitutions par valeurs) *Soient e et e' tels que e' simule e. Alors pour toute valeur v, $e'[v/x]$ simule $e[v/x]$.*

$$\frac{e \leq_{\mathrm{B}} e'}{e[v/x] \leq_{\mathrm{B}} e'[v/x]} \quad (v \in \boldsymbol{V})\,.$$

Démonstration
Considérons un environnement γ. On a :

$$\begin{array}{rcll} \mathrm{Obs}(e[v/x])(\gamma) & = & \mathrm{Obs}^0((e[v/x])[\gamma]) & \text{(prop. 2.2.3 (p. 142))} \\ & = & \mathrm{Obs}^0(e[\gamma.(x : v)]) & \\ & = & \mathrm{Obs}(e)(\gamma.(x : v)) & \text{(prop. 2.2.3 (p. 142))} \\ & \leq & \mathrm{Obs}(e')(\gamma.(x : v)) & \text{(hyp.)} \\ & \leq & \mathrm{Obs}(e'[v/x])(\gamma) & \text{(idem),} \end{array}$$

ce qui montre que $e'[v/x]$ simule $e[v/x]$.
⌞

Ensuite, voyons la compatibilité de l'application à une valeur avec la similarité.

2.2.17 Lemme (Similarité : compatibilité de l'application à une valeur) *Soient e et e' deux termes tels que e' simule e, et v une valeur. Alors $e'\,v$ simule $e\,v$.*

$$\frac{e \leq_{\mathrm{B}} e'}{e\,v \leq_{\mathrm{B}} e'\,v} \quad (v \in \boldsymbol{V})\,.$$

Démonstration
Supposons $e \leq_{\mathrm{B}} e'$.
Soient v une valeur et γ un environnement.
Si $e[\gamma]$ diverge, alors $(e\,v)[\gamma]$ diverge et $\mathrm{Obs}(e\,v)(\gamma) \leq \mathrm{Obs}(e'\,v)(\gamma)$.

Sinon, on a $\mathrm{Obs}(e)(\gamma) = \top\ (\mathrm{Obs}(e\,v)(\gamma))_v$ et $\mathrm{Obs}(e')(\gamma) = \top\ (\mathrm{Obs}(e'\,v)(\gamma))_v$, et $\mathrm{Obs}(e\,v)(\gamma) \leq \mathrm{Obs}(e\,v)(\gamma)$ est une des prémisses du jugement valide $\mathrm{Obs}(e)(\gamma) \leq \mathrm{Obs}(e')(\gamma)$.

⌞

Enfin, étudions la composition de la similarité avec l'équivalence opérationnelle.

2.2.18 Lemme (Similarité : stabilité par composition avec l'équivalence opérationnelle)

Soient e, a, e' et a' quatre termes tels que e' simule e, e et a soient équivalents opérationnellement, de même que e' et a'. Alors a' simule a.

$$(e =_{\mathrm{op}} a)\quad \frac{e \leq_{\mathrm{B}} e'}{a \leq_{\mathrm{B}} a'}\quad (e' =_{\mathrm{op}} a')\,.$$

Démonstration

Soit R la relation entre termes définie par :

$$a\ R\ a' \overset{def}{\Leftrightarrow} \exists\, e, e'\,.\ e \leq_{\mathrm{B}} e' \wedge e =_{\mathrm{op}} a \wedge e' =_{\mathrm{op}} a'\,.$$

Montrons que c'est une simulation.
Soit (a, a') dans R, auquel on associe (e, e') par la définition de R.
Considérons un environnement γ. Si $a[\gamma]$ converge, alors $e[\gamma]$ converge par équivalence opérationnelle, donc $e'[\gamma]$ par similarité, puis $a'[\gamma]$ par équivalence opérationnelle.
Considérons une valeur v. Par le lemme précédent, on a $e\,v \leq_{\mathrm{B}} e'\,v$, et par les propriétés de l'équivalence opérationnelle, $e\,v =_{\mathrm{op}} a\,v$ et $e'\,v =_{\mathrm{op}} a'\,v$. Finalement $(a\,v, a'\,v)$ appartient bien à R.
R est donc une relation de similarité, et par la proposition 2.2.12 (p. 149), on déduit le résultat annoncé.

⌞

Ces lemmes vont nous servir à démontrer les deux propriétés intéressantes de $\leq_{\mathrm{B}}{}^{\bullet}$, sa stabilité par réduction et la préservation de la convergence.

2.2.19 Proposition (Similarité : extension et réduction)

La relation $\leq_{\mathrm{B}}{}^{\bullet}$ vérifie les propriétés suivantes :

– *les substitutions par valeurs passent au quotient par $\leq_{\mathrm{B}}{}^{\bullet}$:*

$$\frac{e \leq_{\mathrm{B}}{}^{\bullet} e' \quad v \leq_{\mathrm{B}}{}^{\bullet} v'}{e[v/x] \leq_{\mathrm{B}}{}^{\bullet} e'[v'/x]}\quad (v, v' \in \boldsymbol{V})\,,$$

– *l'extension de la similarité est stable par réduction à gauche :*

$$\frac{e \leq_{\mathrm{B}}^{\bullet} e'}{e_1 \leq_{\mathrm{B}}^{\bullet} e'} \quad (e \xrightarrow{\mathrm{r}} e_1)\,.$$

– *la convergence est préservée par l'extension de la similarité : si $e \leq_{\mathrm{B}}^{\bullet} e'$ et e converge, alors e' converge.*

Démonstration

• Passage au quotient des substitutions par valeurs

On montre par récurrence structurelle sur e que si $e \leq_{\mathrm{B}}^{\bullet} e'$ et $v \leq_{\mathrm{B}}^{\bullet} v'$ (pour deux valeurs v et v'), alors $e[v/x] \leq_{\mathrm{B}}^{\bullet} e'[v'/x]$.

◦ $e = x$

Par définition de $x \leq_{\mathrm{B}}^{\bullet} e'$, on a $x \leq_{\mathrm{B}} e'$,
puis par le lemme 2.2.16 (p. 153), $v' \leq_{\mathrm{B}} e'[v'/x]$.
Comme $v \leq_{\mathrm{B}}^{\bullet} v'$ par hypothèse, on déduit de la propriété $(\leq_{\mathrm{B}}^{\bullet} \circ \leq_{\mathrm{B}}) \subseteq \leq_{\mathrm{B}}^{\bullet}$ que $v \leq_{\mathrm{B}}^{\bullet} e'[v'/x]$.

◦ $e = y \quad (y \neq x)$

On a $y \leq_{\mathrm{B}} e'$, puis par le lemme 2.2.16 (p. 153), $y[v'/x] \leq_{\mathrm{B}} e'[v'/x]$,
soit $y \leq_{\mathrm{B}} e'[v'/x]$, donc par définition, $y \leq_{\mathrm{B}}^{\bullet} e'[v'/x]$.

◦ $e = \lambda y\, e_1$

Par définition de $\lambda y\, e_1 \leq_{\mathrm{B}}^{\bullet} e'$, il existe e'_1 tel que $\lambda y\, e'_1 \leq_{\mathrm{B}} e'$ et $e_1 \leq_{\mathrm{B}}^{\bullet} e'_1$.
Puis, d'une part :

$$(\lambda y\, e'_1)[v'/x] \quad \leq_{\mathrm{B}} \quad e'[v'/x] \quad \text{(lem. 2.2.16 (p. 153))},$$

d'autre part :

$$\begin{aligned} e_1[v/x] \quad &\leq_{\mathrm{B}}^{\bullet} \quad e'_1[v'/x] \quad \text{(hyp. de réc.)}, \\ \lambda y\, e_1[v/x] \quad &\leq_{\mathrm{B}}^{\bullet} \quad \lambda y\, e'_1[v'/x] \quad \text{(congruence)}. \end{aligned}$$

Comme $(\leq_{\mathrm{B}}^{\bullet} \circ \leq_{\mathrm{B}}) \subseteq \leq_{\mathrm{B}}^{\bullet}$, on déduit des troisième et première inégalités précédentes que $(\lambda y\, e_1)[v/x] \leq_{\mathrm{B}}^{\bullet} e'[v'/x]$.

◦ $e = e_1\, e_2$

Par définition de $e_1\, e_2 \leq_{\mathrm{B}}^{\bullet} e'$, il existe e'_1 et e'_2 tels que $e'_1\, e'_2 \leq_{\mathrm{B}} e'$, $e_1 \leq_{\mathrm{B}}^{\bullet} e'_1$ et $e_2 \leq_{\mathrm{B}}^{\bullet} e'_2$.
De l'hypothèse de récurrence, on déduit $e_1[v/x] \leq_{\mathrm{B}}^{\bullet} e'_1[v'/x]$ et $e_2[v/x] \leq_{\mathrm{B}}^{\bullet} e'_2[v'/x]$, du lemme 2.2.16 (p. 153), $e'_1[v'/x]\, e'_2[v'/x] \leq_{\mathrm{B}} e'[v'/x]$.
On en déduit que $e[v/x] \leq_{\mathrm{B}}^{\bullet} e'[v'/x]$.

• Stabilité par réduction à gauche

On montre par récurrence structurelle sur le contexte de réduction E que si

$E[r] \leq_{\mathrm{B}}^{\bullet} e'$ et $E[r] \xrightarrow{\mathrm{r}} E[r']$ (pour un radical r), alors $E[r'] \leq_{\mathrm{B}}^{\bullet} e'$.
∘ $E = -, r = (\lambda x\, b)\, v, r' = b[v/x]$
Comme $(\lambda x\, b)\, v \leq_{\mathrm{B}}^{\bullet} e'$, il existe f, a et b' tels que

$$\begin{aligned} \lambda x\, b &\leq_{\mathrm{B}}^{\bullet} f\,, \\ v &\leq_{\mathrm{B}}^{\bullet} a\,, \\ f\, a &\leq_{\mathrm{B}} e'\,, \\ b &\leq_{\mathrm{B}}^{\bullet} b'\,, \\ \lambda x\, b' &\leq_{\mathrm{B}} f\,. \end{aligned}$$

De $v \leq_{\mathrm{B}}^{\bullet} a$, il vient que a simule une abstraction, donc converge vers une certaine valeur v', qui vérifie $a =_{\mathrm{B}} v'$. On en déduit que $v \leq_{\mathrm{B}}^{\bullet} v'$. Comme $f\, v' =_{\mathrm{op}} f\, a$, du lemme 2.2.18 (p. 154) appliqué à $f\, a \leq_{\mathrm{B}} e'$, on obtient $f\, v' \leq_{\mathrm{B}} e'$.
D'après la propriété précédente, $b[v/x] \leq_{\mathrm{B}}^{\bullet} b'[v'/x]$. Puis :

$$\begin{aligned} b'[v'/x] &=_{\mathrm{B}} (\lambda x\, b')\, v'\,, \\ (\lambda x\, b')\, v' &\leq_{\mathrm{B}} f\, v' \quad \text{(lem. 2.2.17 (p. 153), } \lambda x\, b' \leq_{\mathrm{B}} f)\,, \\ f\, v' &\leq_{\mathrm{B}} e'\,. \end{aligned}$$

Finalement, par transitivité, $b'[v'/x] \leq_{\mathrm{B}} e'$,
puis par la propriété $(\leq_{\mathrm{B}}^{\bullet} \circ \leq_{\mathrm{B}}) \subseteq \leq_{\mathrm{B}}^{\bullet}$, $b[v/x] \leq_{\mathrm{B}}^{\bullet} e'$.
∘ $E = v\, E_1, E[r] \xrightarrow{\mathrm{r}} E[r']$
Comme $v\, E_1[r] \leq_{\mathrm{B}}^{\bullet} e'$, il existe f et a tels que

$$f\, a \leq_{\mathrm{B}} e'\,, \tag{2.4}$$

$$E_1[r] \leq_{\mathrm{B}}^{\bullet} a\,, \tag{2.5}$$

$$v \leq_{\mathrm{B}}^{\bullet} f\,. \tag{2.6}$$

Par hypothèse de récurrence, $E_1[r'] \leq_{\mathrm{B}}^{\bullet} a$, et comme $\leq_{\mathrm{B}}^{\bullet}$ est une congruence, $v\, E_1[r'] \leq_{\mathrm{B}}^{\bullet} f\, a$. De la propriété $(\leq_{\mathrm{B}}^{\bullet} \circ \leq_{\mathrm{B}}) \subseteq \leq_{\mathrm{B}}^{\bullet}$, on déduit $v\, E_1[r'] \leq_{\mathrm{B}}^{\bullet} e'$.
∘ $E = E_1\, a, E[r] \xrightarrow{\mathrm{r}} E[r']$
Le raisonnement est analogue au précédent.
• Préservation de la convergence
Supposons $e \leq_{\mathrm{B}}^{\bullet} e'$ et que e converge vers $\lambda x\, a$. D'après la propriété précédente, $\lambda x\, a \leq_{\mathrm{B}}^{\bullet} e'$; par définition de l'extension, il s'ensuit que e' simule une abstraction, et donc converge.
⌞

Cette proposition nous permet de montrer les deux conditions manquantes mentionnées en introduction, [ENV] et [RED]. Nous pouvons donc conclure.

2.2.20 Théorème (Bisimilarité et équivalence contextuelle)
Soient e et e' deux termes.

- *e' simule e si et seulement si e converge contextuellement moins que e'.*

$$e \leq_{\mathrm{B}} e' \Leftrightarrow e \leq_{\mathrm{C}} e' .$$

- *e et e' sont bisimilaires si et seulement si e et e' sont équivalents contextuellement.*

$$e =_{\mathrm{B}} e' \Leftrightarrow e =_{\mathrm{C}} e' .$$

Démonstration
C'était l'objet de la proposition 2.2.13 (p. 150) de montrer que le pré-ordre et l'équivalence contextuels étaient inclus dans les relations de similarité et de bisimilarité respectivement. Voyons donc la réciproque.
La relation $\leq_{\mathrm{B}}^{\bullet}$ vérifie :

- $\leq_{\mathrm{B}} \subseteq \leq_{\mathrm{B}}^{\bullet}$ (proposition 2.2.15 (p. 152)),
- $\leq_{\mathrm{B}}^{\bullet}$ est une pré-congruence (proposition 2.2.15),
- si $e \leq_{\mathrm{B}}^{\bullet} e'$, alors pour tout environnement γ, par passage au quotient des substitutions par valeurs (proposition 2.2.19 (p. 154)), $e[\gamma] \leq_{\mathrm{B}}^{\bullet} e'[\gamma]$, puis par préservation de la convergence (proposition 2.2.19), la convergence de $e[\gamma]$ implique celle de $e'[\gamma]$.

L'extension $\leq_{\mathrm{B}}^{\bullet}$ est ainsi une simulation et par la proposition 2.2.12 (p. 149), on déduit que $\leq_{\mathrm{B}}^{\bullet} \subseteq \leq_{\mathrm{B}}$, soit finalement que $\leq_{\mathrm{B}} = \leq_{\mathrm{B}}^{\bullet}$. La relation de similarité est ainsi une pré-congruence.
Supposons que e' simule e, et soit $C_{[\ ]}$ un contexte faisant de $C[e]$ et $C[e']$ des programmes. Par congruence, $C[e']$ simule $C[e]$, et donc si $C[e]$ converge, alors $C[e']$ converge. On a ainsi montré que e converge contextuellement moins que e'.
Par symétrie, on déduit que la bisimilarité est incluse dans l'équivalence contextuelle.
⌞

2.2.3 La sémantique observationnelle

Il est désormais possible de donner une sémantique dénotationnelle à notre langage, à partir des observations : cette sémantique, équivalente au modèle des termes, est complètement adéquate. L'objectif fixé initialement est ainsi rempli. Plutôt que seulement construire cette sémantique à partir des observations, nous allons adopter une approche axiomatique : seront ainsi mises en évidence des conditions qui déterminent de manière unique (à un isomorphisme près) cette sémantique, qu'on qualifiera d'observationnelle.

L'intérêt de cette axiomatisation tient au nombre réduit des axiomes et à leur caractère naturel.

Au préalable, une définition précise des sémantiques dénotationnelles s'impose. Par la suite, on considère la signature $\mathcal{A}$, formée d'un unique symbole fonctionnel d'arité deux, map, représentant l'application. Pour simplifier, toute interprétation de map sera également notée map, ce qui pratiquement ne posera pas de problèmes. Une *$\mathcal{A}$-algèbre ordonnée pointée*[27] est définie par

- un ensemble D et un élément $\perp$ n'appartenant pas à D, dont la réunion $D \cup \perp$ est notée $D_\perp$,
- une relation d'ordre $\leq$, définie sur $D_\perp$ et admettant pour plus petit élément $\perp$,
- une application map : $D_\perp \times D_\perp \to D_\perp$, croissante et stricte en ses deux arguments.

Enfin, un *D-environnement* est une application de l'ensemble des variables dans l'ensemble D. Étant donné un D-environnement Γ, une variable x et un élément d de D, on note $\Gamma.(x : d)$ l'environnement obtenu en modifiant la liaison de x ainsi :

$$(\Gamma.(x : d))(y) \stackrel{def}{=} \begin{cases} \Gamma(y) & \text{si } y \neq x\,, \\ d & \text{sinon.} \end{cases}$$

2.2.21 Définition (Sémantique dénotationnelle)
Une sémantique dénotationnelle *du λ-calcul paresseux avec appel par valeurs est définie par :*

- *une $\mathcal{A}$-algèbre ordonnée pointée, $(D_\perp, \leq, \mathrm{map})$, dont le domaine définit l'ensemble des dénotations,*
- *une interprétation qui à tout terme e et à tout D-environnement Γ, associe une dénotation dans $D_\perp$, notée $[\![e]\!]_\Gamma$,*

vérifiant trois propriétés :

- [EXT] (extensionalité de $(D_\perp, \leq, \mathrm{map})$)

$$\forall\, f, f' \in D\,.\ (\forall\, d \in D\,.\ \mathrm{map}(f, d) \leq \mathrm{map}(f', d) \Rightarrow f \leq f')\,,$$

- [REC] (compositionalité)

[27] L'usage est de qualifier une algèbre en décrivant son domaine, ses opérations devant alors vérifier une condition naturelle associée à ces qualificatifs.

l'interprétation vérifie les équations récursives suivantes :

$$\begin{aligned} [\![x]\!]_\Gamma &= \Gamma(x)\,, \\ \mathrm{map}([\![\,\lambda x\, e]\!]_\Gamma, d) &= [\![e]\!]_{\Gamma.(x:d)} \quad (d \in D)\,, \\ [\![e_1\, e_2]\!]_\Gamma &= \mathrm{map}([\![e_1]\!]_\Gamma, [\![e_1]\!]_\Gamma)\,, \end{aligned}$$

– [VAL] (correction des dénotations de valeurs)
l'interprétation d'une valeur est une dénotation non triviale :

$$\forall\, v \in \boldsymbol{V}.\ \forall\, \Gamma \in D^{\boldsymbol{X}}.\ [\![v]\!]_\Gamma \neq \bot\,.$$

Étant donné une telle sémantique dénotationnelle, on définit sur les termes un pré-ordre, $\leq_{\mathrm{D}}$, et la relation d'équivalence associée, $=_{\mathrm{D}}$, par

$$\begin{aligned} e \leq_{\mathrm{D}} e' &\stackrel{def}{\Leftrightarrow} \forall\, \Gamma \in D^{\boldsymbol{X}}.\ [\![e]\!]_\Gamma \leq [\![e']\!]_\Gamma\,, \\ e =_{\mathrm{D}} e' &\stackrel{def}{\Leftrightarrow} e \leq_{\mathrm{D}} e' \wedge e' \leq_{\mathrm{D}} e\,. \end{aligned}$$

Pour se convaincre de la pertinence de cette définition, montrons quelques lemmes classiques, et bien utiles par la suite. Pour la sémantique dénotationnelle considérée, on reprend les notations de la définition 2.2.21 ; par environnement, on entend D-environnement.
Tout d'abord, l'interprétation d'un terme ne dépend de l'environnement que par ses variables libres.

2.2.22 Lemme
Soit e un terme. Considérons deux environnement Γ et Γ' tels que pour toute variable libre x de e, on ait $\Gamma(x) = \Gamma'(x)$. Alors

$$[\![e]\!]_\Gamma = [\![e]\!]_{\Gamma'}\,.$$

Démonstration
À chaque terme e, on associe la relation d'équivalence entre environnements $\equiv_e$, définie par :

$$\Gamma \equiv_e \Gamma' \stackrel{def}{\Leftrightarrow} \forall\, x \in \mathbf{FV}(e)\,.\ \Gamma(x) = \Gamma'(x)\,.$$

On montre par récurrence structurelle sur e que $\forall\, \Gamma, \Gamma'.\ \Gamma \equiv_e \Gamma' \Rightarrow [\![e]\!]_\Gamma = [\![e]\!]_{\Gamma'}$. Pour chaque cas de la récurrence structurelle sur e, on considère deux environnements Γ et Γ', vérifiant $\Gamma \equiv_e \Gamma'$.

• $e = x$
On a :

$$\begin{aligned} [\![x]\!]_\Gamma &= \Gamma(x) \\ &= \Gamma'(x) \quad (x \in \mathbf{FV}(e)) \\ &= [\![x]\!]_{\Gamma'} . \end{aligned}$$

• $e = \lambda x\, a$
On a, pour tout d de D :

$$\begin{aligned} \mathrm{map}([\![\lambda x\, a]\!]_\Gamma, d) &= [\![a]\!]_{\Gamma.(x:d)} \\ &= [\![a]\!]_{\Gamma'.(x:d)} \quad \text{(hyp. de réc.)} \\ &= \mathrm{map}([\![\lambda x\, a]\!]_{\Gamma'}, d) . \end{aligned}$$

On conclut à l'égalité de $[\![\lambda x\, a]\!]_\Gamma$ et $[\![\lambda x\, a]\!]_{\Gamma'}$ par extensionalité ([EXT]), comme $[\![\lambda x\, a]\!]_\Gamma$ et $[\![\lambda x\, a]\!]_{\Gamma'}$ appartiennent à D (d'après [VAL]).
• $e = e_1\, e_2$
On a :

$$\begin{aligned} [\![e_1\, e_2]\!]_\Gamma &= \mathrm{map}([\![e_1]\!]_\Gamma, [\![e_2]\!]_\Gamma) \\ &= \mathrm{map}([\![e_1]\!]_{\Gamma'}, [\![e_2]\!]_{\Gamma'}) \quad \text{(hyp. de réc.)} \\ &= [\![e_1\, e_2]\!]_{\Gamma'} . \end{aligned}$$

⌞

En particulier, l'interprétation d'un programme e ne dépend pas de l'environnement d'interprétation : on notera $[\![e]\!]_\emptyset$ cette dénotation constante.
Substituer une valeur à une variable revient à la lier dans l'environnement à l'interprétation de la valeur.

2.2.23 Lemme (Substitution par valeur et liaison dans l'environnement)
Soient e un terme, v une valeur et Γ un environnement. Alors :

$$[\![e[v/x]]\!]_\Gamma = [\![e]\!]_{\Gamma.(x:[\![v]\!]_\emptyset)} .$$

Démonstration
On montre le résultat par récurrence structurelle sur e. Les cas $e = y$, $e = x$ et $e = e_1\, e_2$ se déduisent directement de la définition inductive de l'interprétation ([REC]). Le seul cas intéressant est donc $e = \lambda y\, a$ (avec

$y \neq x$).
On a, pour tout d de D :

$$\begin{aligned}
\mathrm{map}(\llbracket(\lambda y\, a)[v/x]\rrbracket_\Gamma, d) &= \mathrm{map}(\llbracket\, \lambda y\, a[v/x]\rrbracket_\Gamma, d) \\
&= \llbracket a[v/x]\rrbracket_{\Gamma.(y\,:\,d)} \\
&= \llbracket a\rrbracket_{\Gamma.(x\,:\,\llbracket v\rrbracket_\emptyset).(y\,:\,d)} \quad \text{(hyp. de réc.)} \\
&= \mathrm{map}(\llbracket\, \lambda y\, a\rrbracket_{\Gamma.(x\,:\,\llbracket v\rrbracket_\emptyset)}, d)\,.
\end{aligned}$$

On conclut à l'égalité de $\llbracket(\lambda y\, a)[v/x]\rrbracket_\Gamma$ et $\llbracket\, \lambda y\, a\rrbracket_{\Gamma.(x\,:\,\llbracket v\rrbracket_\emptyset)}$ par extensionalité ([EXT]), comme ces deux dénotations appartiennent à D (d'après [VAL]).
⌞

Ainsi, les substitutions par valeurs passent au quotient par l'équivalence dénotationnelle $=_{\mathrm{D}}$.
L'équivalence dénotationnelle forme une congruence, ce qu'exprime ce lemme de manière un peu plus générale en considérant le pré-ordre dénotationnel.

2.2.24 Lemme (Pré-congruence de $\leq_{\mathbf{D}}$)
Le pré-ordre dénotationnel forme une pré-congruence, et l'équivalence dénotationnelle une congruence.

Démonstration
On vérifie la stabilité de $\leq_{\mathrm{D}}$ pour les règles [CONG-VAR], [CONG-ABS] et [CONG-APP] (cf. définition 2.2.8 (p. 146)).
• [CONG-VAR]
Immédiat.
• [CONG-ABS]
Supposons $e_1 \leq_{\mathrm{D}} e_2$. On a :

$$\begin{aligned}
\mathrm{map}(\llbracket\, \lambda x\, e_1\rrbracket_\Gamma, d) &= \llbracket e_1\rrbracket_{\Gamma.(x\,:\,d)} \\
&\leq \llbracket e_2\rrbracket_{\Gamma.(x\,:\,d)} \quad \text{(hyp.)} \\
&= \mathrm{map}(\llbracket\, \lambda x\, e_2\rrbracket_\Gamma, d)\,.
\end{aligned}$$

On conclut que $\llbracket\, \lambda x\, e_1\rrbracket_\Gamma \leq \llbracket\, \lambda x\, e_2\rrbracket_\Gamma$ par extensionalité ([EXT]), comme ces deux dénotations appartiennent à D (d'après [VAL]), ce qui prouve que $\lambda x\, e_1 \leq_{\mathrm{D}} \lambda x\, e_2$.
• [CONG-APP]
C'est immédiat par croissance de map.
⌞

Il est maintenant possible de préciser le rapport entre la sémantique opérationnelle et la sémantique dénotationnelle.

2.2.25 Lemme (Stabilité des classes modulo l'équivalence dénotationnelle par évaluation)

Supposons que e converge vers v. Alors :

$$[\![e]\!]_\emptyset = [\![v]\!]_\emptyset \,.$$

Démonstration

Montrons que si e_1 se réduit en e_2, alors $[\![e_1]\!]_\emptyset = [\![e_2]\!]_\emptyset$, en procédant par récurrence structurelle sur le contexte d'évaluation. On conclut ensuite par récurrence sur la longueur de la trace.

On considère la réduction $E[(\lambda x\, a)\, v] \xrightarrow{r} E[a[v/x]]$.

- $E = -$

Soit Γ un environnement quelconque. On a successivement :

$$\begin{aligned} [\![(\lambda x\, a)\, v]\!]_\Gamma &= \mathrm{map}([\![\lambda x\, a]\!]_\Gamma, [\![v]\!]_\Gamma) \quad ([\mathrm{REC}]) \\ &= [\![a]\!]_{\Gamma.(x\,:\,[\![v]\!]_\Gamma)} \quad ([\mathrm{REC}]) \\ &= [\![a[v/x]]\!]_\Gamma \quad (\text{lem. 2.2.23 (p. 160)}) \,. \end{aligned}$$

- $E = E'\, e$ ou $E = v\, E'$

Le résultat découle par congruence (lemme 2.2.24 (p. 161)) de l'hypothèse de récurrence.

⌞

On peut en déduire la préservation de la convergence.

2.2.26 Corollaire (Préservation de la convergence)

L'interprétation dénotationnelle préserve la convergence :

– [CONV] *(préservation de la convergence)*

$$\forall\, e \in \Lambda^0 \,.\; \neg(e \Downarrow \bot) \Rightarrow [\![e]\!]_\emptyset \neq \bot \,.$$

Démonstration

Si le programme e converge vers v, alors par le lemme 2.2.25, $[\![e]\!]_\emptyset = [\![v]\!]_\emptyset$. Comme v est une valeur, par [VAL], $[\![v]\!]_\emptyset \neq \bot$, soit $[\![e]\!]_\emptyset \neq \bot$.

⌞

La réciproque est une propriété éminemment souhaitable : en effet, elle implique que la sémantique dénotationnelle est *adéquate*.

2.2.27 Proposition (Adéquation de la sémantique dénotationnelle)

Supposons que l'interprétation dénotationnelle préserve la divergence :

– [DIV] *(préservation de la divergence)*

$$\forall\, e \in \Lambda^0\,.\; e \Downarrow \bot \Rightarrow [\![e]\!]_\emptyset = \bot\,.$$

Alors la sémantique dénotationnelle est adéquate *:*
le pré-ordre dénotationnel $\leq_{\mathrm{D}}$ *est inclus dans le pré-ordre contextuel* $\leq_{\mathrm{C}}$*, tout comme l'équivalence dénotationnelle* $=_{\mathrm{D}}$ *dans l'équivalence contextuelle* $=_{\mathrm{C}}$*.*

Démonstration
Supposons [DIV].
Soient deux termes e et e' vérifiant $e \leq_{\mathrm{D}} e'$, et soit $C_{[\ ]}$ un contexte faisant de e et de e' des programmes.
On montre que si $C[e']$ diverge, alors $C[e]$ diverge, ce qui nous permettra de conclure que $e \leq_{\mathrm{C}} e'$.
Supposons donc que $C[e']$ diverge.
Comme $\leq_{\mathrm{D}}$ est une pré-congruence, $C[e] \leq_{\mathrm{D}} C[e']$. Par [DIV], on a $[\![C[e']]\!]_\emptyset = \bot$, puis par majoration, $[\![C[e]]\!]_\emptyset = \bot$, dont on déduit par [CONV] en contraposant que $C[e]$ diverge.
Par symétrie, on déduit le résultat pour l'équivalence dénotationnelle.
⌞

Venons-en maintenant à la construction d'une sémantique dénotationnelle à partir des observations.
Soient $\mathbb{O}$ l'ensemble des observations non triviales :

$$\mathbb{O} \stackrel{def}{=} \mathrm{Obs}^0(\mathbf{V})\,,$$

et $\mathbb{O}_\bot$ l'ensemble des observations :

$$\mathbb{O}_\bot \stackrel{def}{=} \{\bot\} \cup \mathbb{O}\,.$$

Ce sont les observations qui vont servir de dénotations. La définition de la sémantique observationnelle va être paramétrée par une application vérifiant certaines propriétés, cependant nous montrerons qu'elle est indépendante du choix de l'application. On supposera en effet la donnée d'une application ρ définie sur l'ensemble des observations $\mathbb{O}_\bot$ et vérifiant :

– $\rho(\bot)$ est un programme divergent,
– pour toute observation non triviale d (différente de $\bot$), $\rho(d)$ est une valeur vérifiant

$$\mathrm{Obs}^0(\rho(d)) = d\,.$$

On en déduit que pour toute observation d, on a $\mathrm{Obs}^0(\rho(d)) = d$. Aussi, ρ est croissante et injective, puisqu'on a l'équivalence suivante, pour tout couple d'observations d_1 et d_2 :

$$d_1 \leq d_2 \Leftrightarrow \rho(d_1) \leq_{\mathrm{B}} \rho(d_1)\,.$$

Autre propriété importante que nous utiliserons : si e est un programme, alors $\rho(\mathrm{Obs}^0(e))$ et e sont bisimilaires.
Bien sûr, il existe une telle application, puisqu'on peut prendre comme programme divergent $(\lambda x\, x\, x)\,(\lambda x\, x\, x)$ et qu'à toute observation non triviale, il est possible d'associer une valeur vérifiant la propriété par définition.
Grâce à une telle application, l'ensemble des observations, qui est déjà ordonné par l'ordre induit et possède $\bot$ comme plus petit élément, peut être muni d'une opération map, le transformant en une algèbre ordonnée pointée.

2.2.28 Définition et proposition (Ensemble des observations : la structure algébrique)

Soit ρ une application définie sur l'ensemble des observations $\mathbb{O}_\bot$ et vérifiant :

(i) $\rho(\bot)$ est un programme divergent,

(ii) pour toute observation non triviale d (différente de $\bot$), $\rho(d)$ est une valeur vérifiant

$$\mathrm{Obs}^0(\rho(d)) = d\,.$$

L'ensemble ordonné $(\mathbb{O}_\bot, \leq)$ peut être muni d'une structure de $\mathcal{A}$-algèbre ordonnée pointée en définissant l'opération map *ainsi, pour toutes observations f et d de $\mathbb{O}_\bot$:*

$$\mathrm{map}(f, d) \quad = \quad \mathrm{Obs}^0(\rho(f)\,\rho(d))\,.$$

De plus, cette définition ne dépend pas du choix de ρ.

Démonstration
Montrons que $(\mathbb{O}_\bot, \leq, \mathrm{map})$ forme bien une structure de $\mathcal{A}$-algèbre ordonnée pointée.
L'opération map est évidemment stricte en ses deux arguments. Montrons qu'elle est croissante.
Soient f_1, f_2, d_1 et d_2 quatre observations vérifiant $f_1 \leq f_2$ et $d_1 \leq d_2$. Par croissance de ρ, on a $\rho(f_1) \leq_{\mathrm{B}} \rho(f_2)$ et $\rho(d_1) \leq_{\mathrm{B}} \rho(d_2)$, puis par congruence, $\rho(f_1)\,\rho(d_1) \leq_{\mathrm{B}} \rho(f_2)\,\rho(d_2)$, ce qui est équivalent à $\mathrm{Obs}^0(\rho(f_1)\,\rho(d_1)) \leq \mathrm{Obs}^0(\rho(f_2)\,\rho(d_2))$.

Il s'agit enfin de montrer que la définition de map ne dépend pas du choix de ρ. C'est immédiat par congruence de la bisimilarité.
⌞

Cette définition met en évidence le fait que l'ensemble des observations vérifie l'inéquation suivante

$$\mathbb{O} \subset \mathbb{O} \to \mathbb{O}_\perp \,,$$

à un isomorphisme près, à rapprocher de l'équation classique entre domaines

$$X = X \to X_\perp \,,$$

où l'espace fonctionnel est celui des fonctions continues.
Il est maintenant possible de donner une interprétation observationnelle des termes.

2.2.29 Définition et théorème (Sémantique observationnelle)
Soit ρ une application définie sur l'ensemble des observations $\mathbb{O}_\perp$ et vérifiant :

(i) $\rho(\perp)$ est un programme divergent,

(ii) pour toute observation non triviale d (différente de $\perp$), $\rho(d)$ est une valeur vérifiant

$$\mathrm{Obs}^0(\rho(d)) = d \,.$$

L'interprétation observationnelle d'un terme e est une application, notée $[\![e]\!]$, qui associe à tout $\mathbb{O}$-environnement Γ, la dénotation observationnelle suivante, notée $[\![e]\!]_\Gamma$:

$$[\![e]\!]_\Gamma \stackrel{def}{=} \mathrm{Obs}(e)(\rho \circ \Gamma) \,.$$

Alors $((\mathbb{O}_\perp, \leq, \mathrm{map}), [\![-]\!]_-)$ définit une sémantique dénotationnelle, la sémantique observationnelle*, qui de plus, ne dépend pas du choix de ρ.*
À cette sémantique observationnelle, sont associés le pré-ordre observationnel $\leq_\mathrm{O}$ et l'équivalence observationnelle $=_\mathrm{O}$.

Démonstration
On vérifie que la sémantique dénotationnelle vérifie les propriétés [EXT], [REC] et [VAL].
• [EXT]
Soient f et g appartenant à $\mathbb{O}$ tels que pour tout d de $\mathbb{O}$, on ait $\mathrm{map}(f,d) \leq \mathrm{map}(g,d)$, soit $\mathrm{Obs}^0(\rho(f)\,\rho(d)) \leq \mathrm{Obs}^0(\rho(g)\,\rho(d))$.
Comme f et g sont différents de $\perp$, on a $f = \mathrm{Obs}^0(\rho(f)) = \top\ (\mathrm{Obs}^0(\rho(f)\,v))_{v\in\mathbf{V}}$

et $g = \mathrm{Obs}^0(\rho(g)) =\top (\mathrm{Obs}^0(\rho(g)\, v))_{v\in\mathbf{V}}$.
Soit v une valeur. Pour conclure, il suffit de montrer que $\mathrm{Obs}^0(\rho(f)\, v) \leq \mathrm{Obs}^0(\rho(g)\, v)$.
Comme $\rho(\mathrm{Obs}^0(v)) =_{\mathrm{B}} v$, puis par congruence $\rho(f)\,\rho(\mathrm{Obs}^0(v)) =_{\mathrm{B}} \rho(f)\, v$, et de même, $\rho(g)\,\rho(\mathrm{Obs}^0(v)) =_{\mathrm{B}} \rho(g)\, v$, on déduit de l'hypothèse l'inégalité recherchée.
• [REC]
On a, pour la première égalité concernant une variable x :

$$\begin{aligned}[\![x]\!]_\Gamma &= \mathrm{Obs}(x)(\rho\circ\Gamma)\\ &= \mathrm{Obs}^0(x[\rho\circ\Gamma])\\ &= \mathrm{Obs}^0((\rho\circ\Gamma)(x))\\ &= \Gamma(x)\,.\end{aligned}$$

Pour la seconde égalité concernant une abstraction $\lambda x\, e$, on doit montrer que pour tout $d \in \mathbb{O}$, on a :

$$\mathrm{map}([\![\,\lambda x\, e]\!]_\Gamma, d) = [\![e]\!]_{\Gamma.(x\,:\,d)}\,.$$

On a :

$$\begin{aligned}\mathrm{map}([\![\,\lambda x\, e]\!]_\Gamma, d) &= \mathrm{map}(\mathrm{Obs}^0((\lambda x\, e)[\rho\circ\Gamma]), d)\\ &= \mathrm{Obs}^0(\rho(\mathrm{Obs}^0((\lambda x\, e)[\rho\circ\Gamma]))\,\rho(d))\,.\end{aligned}$$

Comme $\rho(\mathrm{Obs}^0((\lambda x\, e)[\rho\circ\Gamma])) =_{\mathrm{B}} (\lambda x\, e)[\rho\circ\Gamma]$, on obtient finalement :

$$\begin{aligned}\mathrm{map}([\![\,\lambda x\, e]\!]_\Gamma, d) &= \mathrm{Obs}^0((\lambda x\, e)[\rho\circ\Gamma]\,\rho(d)) \quad \text{(congruence)}\\ &= \mathrm{Obs}^0(e[(\rho\circ\Gamma).(x\ :\ \rho(d))])\\ &= \mathrm{Obs}^0(e[\rho\circ(\Gamma.(x\ :\ d))])\\ &= [\![e]\!]_{\Gamma.(x\,:\,d)}\,.\end{aligned}$$

Pour la troisième égalité concernant une application $e_1\, e_2$, on a :

$$\begin{aligned}\mathrm{map}([\![e_1]\!]_\Gamma, [\![e_2]\!]_\Gamma) &= \mathrm{map}(\mathrm{Obs}^0(e_1[\rho\circ\Gamma]), \mathrm{Obs}^0(e_2[\rho\circ\Gamma]))\\ &= \mathrm{Obs}^0(\rho(\mathrm{Obs}^0(e_1[\rho\circ\Gamma]))\,\rho(\mathrm{Obs}^0(e_2[\rho\circ\Gamma])))\,.\end{aligned}$$

Comme $\rho(\mathrm{Obs}^0(e_1[\rho\circ\Gamma])) =_{\mathrm{B}} e_1[\rho\circ\Gamma]$, et de même pour e_2, on obtient par congruence

$$\mathrm{map}([\![e_1]\!]_\Gamma, [\![e_2]\!]_\Gamma) = \mathrm{Obs}^0(e_1[\rho\circ\Gamma]\, e_2[\rho\circ\Gamma])$$

soit

$$\mathrm{map}([\![e_1]\!]_\Gamma, [\![e_2]\!]_\Gamma) = [\![e_1\, e_2]\!]_\Gamma\,.$$

• [VAL]
Cette propriété est trivialement vérifiée, par définition de l'observation.
Pour finir, montrons que la définition de la sémantique ne dépend pas du choix de ρ.
Soient ρ_1 et ρ_2 deux applications vérifiant les hypothèses de l'énoncé. On doit montrer que pour tout terme e et tout environnement Γ, on a :

$$\mathrm{Obs}(e)(\rho_1 \circ \Gamma) = \mathrm{Obs}(e)(\rho_2 \circ \Gamma)\,,$$

soit

$$e[\rho_1 \circ \Gamma] =_{\mathrm{B}} e[\rho_2 \circ \Gamma]\,.$$

Comme pour toute variable x, $\rho_1 \circ \Gamma(x) =_{\mathrm{B}} \rho_2 \circ \Gamma(x)$, on peut conclure du fait que la bisimilarité est égale à l'équivalence contextuelle et que les substitutions par valeurs passent au quotient par cette équivalence (cf. prop. 2.2.11 (p. 148)).
⌞

La sémantique observationnelle nous convient bien, l'équivalence observationnelle coïncidant avec l'équivalence contextuelle.

2.2.30 Théorème (Adéquation complète)
La sémantique observationnelle est complètement adéquate *:*

$$\forall\, e, e'\,.\; e \leq_{\mathrm{O}} e' \Leftrightarrow e \leq_{\mathrm{C}} e'\,.$$

Démonstration
D'une part, on a les équivalences suivantes :

$$\begin{array}{rcl} e \leq_{\mathrm{O}} e' & \Leftrightarrow & \forall\, \Gamma\,.\; [\![e]\!]_\Gamma \leq [\![e']\!]_\Gamma \\ & \Leftrightarrow & \forall\, \Gamma\,.\; \mathrm{Obs}(e)(\rho \circ \Gamma) \leq \mathrm{Obs}(e')(\rho \circ \Gamma)\,. \end{array}$$

D'autre part, on a :

$$\begin{array}{rcl} e \leq_{\mathrm{C}} e' & \Leftrightarrow & e \leq_{\mathrm{B}} e' \\ & \Leftrightarrow & \forall\, \gamma\,.\; \mathrm{Obs}(e)(\gamma) \leq \mathrm{Obs}(e')(\gamma)\,. \end{array}$$

Que cette dernière proposition implique $e \leq_{\mathrm{O}} e'$ est donc clair.
Supposons $e \leq_{\mathrm{O}} e'$, et soit γ un environnement opérationnel. Soit Γ l'environnement observationnel défini par $\Gamma(x) = \mathrm{Obs}^0(\gamma(x))$. Pour toute variable x, on a $\rho(\Gamma(x)) =_{\mathrm{B}} \gamma(x)$. Il s'ensuit que $e[\rho \circ \Gamma] =_{\mathrm{B}} e[\gamma]$ et $e'[\rho \circ \Gamma] =_{\mathrm{B}} e'[\gamma]$, soit $\mathrm{Obs}(e)(\rho \circ \Gamma) = \mathrm{Obs}(e)(\gamma)$ et $\mathrm{Obs}(e')(\rho \circ \Gamma) = \mathrm{Obs}(e')(\gamma)$, si bien qu'on

peut conclure, en utilisant l'hypothèse.
⌞

La sémantique observationnelle nous fournit une représentation du modèle des termes. Après sa construction, on peut adopter une démarche axiomatique :

> trouver des conditions définissant (à un isomorphisme près) la sémantique observationnelle.

Ces conditions peuvent porter sur le domaine dénotationnel, l'interprétation dénotationnelle et la sémantique opérationnelle, donnée par la fonction d'évaluation, comme c'était déjà le cas lors de notre définition des sémantiques dénotationnelles (cf. p. 158). Une axiomatisation intéressante doit faciliter la preuve de règles admissibles dans les théories équationnelles et inéquationnelles induites sur les termes par l'équivalence et le pré-ordre contextuels ; rappelons que l'intérêt d'une sémantique complètement adéquate réside dans sa capacité à décrire complètement ces théories. C'est donc la seule pratique qui doit déterminer l'intérêt des axiomes choisis ; pour rester fidèle à la ligne fixée initialement, elle doit s'appuyer sur des techniques opérationnelles, supposées plus proches de l'intuition du programmeur.
Les axiomes retenus reprennent ceux définissant une sémantique dénotationnelle, en renforçant la propriété d'extensionalité. S'y ajoutent la préservation de la divergence opérationnelle, qui implique l'adéquation, et une condition d'accessibilité des dénotations.

2.2.31 Définition (Axiomatisation de la sémantique observationnelle)

Une sémantique observationnelle *du λ-calcul avec appel par valeur est définie par :*

- *une $\mathcal{A}$-algèbre ordonnée pointée, $(D_\perp, \leq, \mathrm{map})$, dont le domaine définit l'ensemble des dénotations,*
- *une interprétation qui à tout terme e et à tout D-environnement Γ, associe une dénotation dans $D_\perp$, notée $[\![e]\!]_\Gamma$,*

vérifiant

- [EXT+] (extensionalité forte de $(D_\perp, \leq, \mathrm{map})$)
  la relation d'ordre $\leq$ est engendrée co-inductivement par le système

suivant :

$$\frac{\emptyset}{(\bot, d)} \quad (d \in D_\bot)\,,$$

$$\frac{(\mathrm{map}(f,d), \mathrm{map}(g,d))_{d\in D}}{(f,g)} \quad (f,g \in D)\,,$$

– [REC] (compositionalité)
l'interprétation vérifie les équations récursives suivantes :

$$\begin{aligned} [\![x]\!]_\Gamma &= \Gamma(x)\,, \\ \mathrm{map}([\![\,\lambda x\, e]\!]_\Gamma, d) &= [\![e]\!]_{\Gamma.(x:d)} \quad (d \in D)\,, \\ [\![e_1\, e_2]\!]_\Gamma &= \mathrm{map}([\![e_1]\!]_\Gamma, [\![e_1]\!]_\Gamma)\,, \end{aligned}$$

– [VAL] (correction des dénotations de valeurs)
l'interprétation d'une valeur est une dénotation non triviale :

$$\forall\, v \in \boldsymbol{V}.\ \forall\, \Gamma \in D^{\boldsymbol{X}}.\ [\![v]\!]_\Gamma \neq \bot\,,$$

– [DIV] (préservation de la divergence)
l'interprétation préserve la divergence :

$$\forall\, e \in \Lambda^0\,.\ e \Downarrow \bot \Rightarrow \forall\, \Gamma \in D^{\boldsymbol{X}}.\ [\![e]\!]_\Gamma = \bot\,,$$

– [ACC] (accessibilité)
chaque élément de D dénote une valeur :

$$\forall\, d \in D\,.\ \exists\, v \in \boldsymbol{V}, \Gamma \in D^{\boldsymbol{X}}.\ [\![v]\!]_\Gamma = d\,.$$

Étant donné une telle sémantique observationnelle, on définit sur les termes un pré-ordre, $\leq_{\mathrm{O}}$, et la relation d'équivalence associée, $=_{\mathrm{O}}$, par

$$\begin{aligned} e \leq_{\mathrm{O}} e' &\stackrel{def}{\Leftrightarrow} \forall\, \Gamma \in D^{\boldsymbol{X}}.\ [\![e]\!]_\Gamma \leq [\![e']\!]_\Gamma\,, \\ e =_{\mathrm{O}} e' &\stackrel{def}{\Leftrightarrow} e \leq_{\mathrm{O}} e' \wedge e' \leq_{\mathrm{O}} e\,. \end{aligned}$$

Une sémantique observationnelle est évidemment dénotationnelle, puisque la condition d'extensionalité forte [EXT+] implique celle d'extensionalité [EXT], qui affirme seulement que la relation d'ordre est stable pour la seconde règle d'inférence.
La condition d'accessibilité permet de ramener les raisonnements sur les termes à des raisonnements sur les programmes, comme nous l'indique ce lemme, forme de réciproque du lemme 2.2.23 (p. 160), concernant la compatibilité des substitutions par valeurs.

2.2.32 Lemme (Pré-ordre dénotationnel : Stabilité par extension opérationnelle)

*Soient e et e' deux termes appartenant à l'*extension opérationnelle *du pré-ordre dénotationnel, c'est-à-dire vérifiant pour tout environnement opérationnel γ :*

$$e[\gamma] \leq_{\mathrm{O}} e'[\gamma]\,.$$

Alors :

$$e \leq_{\mathrm{O}} e'\,.$$

Démonstration

On suppose que pour tout environnement opérationnel γ, on ait $e[\gamma] \leq_{\mathrm{O}} e'[\gamma]$.

Soit Γ un environnement dénotationnel quelconque. On doit montrer que $[\![e]\!]_\Gamma \leq [\![e']\!]_\Gamma$.

Par la condition d'accessibilité [ACC], il existe un environnement opérationnel γ vérifiant $\forall\, x\,.\ \Gamma(x) = [\![\gamma(x)]\!]_\emptyset$. Par le lemme 2.2.23 (p. 160), on a $[\![e]\!]_\Gamma = [\![e[\gamma]]\!]_\emptyset$ et $[\![e']\!]_\Gamma = [\![e'[\gamma]]\!]_\emptyset$. Par hypothèse, $[\![e[\gamma]]\!]_\emptyset \leq [\![e'[\gamma]]\!]_\emptyset$, d'où l'inégalité recherchée.

⌞

Par la suite, nous utiliserons souvent ce lemme ainsi, ou sous une forme voisine, sans le rappeler : étant donné deux termes e et e' vérifiant une certaine condition $H(e, e')$, on cherche à montrer que $F(e) \leq_{\mathrm{O}} G(e')$, où $F(e)$ et $G(e')$ sont deux termes calculés à partir de e et e' respectivement ; si pour tout environnement opérationnel γ, la condition $H(e[\gamma], e'[\gamma])$ est vérifiée dès lors que $H(e, e')$ l'est, et si F et G commutent avec γ (soit $F(e)[\gamma] = F(e[\gamma]), G(e')[\gamma] = G(e'[\gamma])$), alors il est possible de se limiter dans la démonstration au cas où e et e' sont des programmes.

La sémantique observationnelle construite précédemment vérifie évidemment toutes les conditions de la définition et nous assure de l'existence d'au moins une sémantique dénotationnelle vérifiant ces conditions. Pour montrer l'unicité, prouvons un résultat important, l'*adéquation complète*.

2.2.33 Théorème (Adéquation complète)

Considérons une sémantique observationnelle (suivant la définition 2.2.31 (p. 168)). Alors elle est complètement adéquate *:*

$$\forall\, e, e' \in \Lambda\,.\ e \leq_{\mathrm{O}} e' \Leftrightarrow e \leq_{\mathrm{C}} e'\,.$$

Démonstration

L'adéquation découle de la proposition 2.2.27 (p. 162). Il reste à montrer

qu'elle est complète.
Soient e et e' deux termes tels que $e \leq_{\mathrm{C}} e'$. Comme par la proposition 2.2.11 (p. 148), pour tout environnement opérationnel γ, $e[\gamma] \leq_{\mathrm{C}} e'[\gamma]$, on peut se limiter au cas où e et e' sont des programmes, ce qu'on suppose.
Par [ACC], à tout élément d de D, il est possible d'associer une valeur notée v_d telle que $[\![v_d]\!]_\emptyset = d$. Remarquons que pour tout programme e, on a $[\![e\, v_d]\!]_\emptyset = \mathrm{map}([\![e]\!]_\emptyset, d)$.
L'inégalité $e \leq_{\mathrm{O}} e'$ est équivalente à l'inégalité $[\![e]\!]_\emptyset \leq [\![e']\!]_\emptyset$, qu'on va montrer en la prouvant dans le système d'inférence défini par [EXT+].
Pour tout couple de programmes (e, e') tel que $e \leq_{\mathrm{C}} e'$, on construit une preuve de $[\![e]\!]_\emptyset \leq [\![e']\!]_\emptyset$ en résolvant le système d'équations récursives, d'inconnues $(X_{(e,e')})_{e,e' \in \Lambda^0 | e \leq_{\mathrm{C}} e'}$, et défini par les équations :

$$X_{(e,e')} = \begin{cases} \dfrac{\emptyset}{([\![e]\!]_\emptyset, [\![e']\!]_\emptyset)} & \text{si } e \text{ diverge,} \\[2ex] \dfrac{(X_{(e\, v_d, e'\, v_d)})_{d \in D}}{([\![e]\!]_\emptyset, [\![e']\!]_\emptyset)} & \text{si } e \text{ converge.} \end{cases}$$

Dans le premier cas, si e diverge, on a d'après [DIV], $[\![e]\!]_\emptyset = \bot$, et le jugement est un axiome.
Dans le second cas, si e converge, alors e' aussi, puisque e' converge contextuellement plus que e ; par [CONV] (cf. corollaire 2.2.26 (p. 162)), $[\![e]\!]_\emptyset$ et $[\![e']\!]_\emptyset$ appartiennent à D ; de plus, la valeur de la solution en $X_{(e\, v_d, e'\, v_d)}$ a pour racine $([\![e\, v_d]\!]_\emptyset, [\![e'\, v_d]\!]_\emptyset)$, soit $(\mathrm{map}([\![e]\!]_\emptyset, d), \mathrm{map}([\![e']\!]_\emptyset, d))$.
Il en résulte que ce système quasi-uniforme est compatible avec le système d'inférence défini par [EXT+]. D'après la proposition 1.2.3 (p. 73), la valeur de l'unique solution du système en (e, e') constitue une preuve de $[\![e]\!]_\emptyset \leq [\![e']\!]_\emptyset$, ce qui montre que $e \leq_{\mathrm{O}} e'$.
⌞

Avant de montrer que l'axiomatisation détermine une unique sémantique observationnelle à un isomorphisme près, précisons cette notion d'équivalence entre sémantiques dénotationnelles.

2.2.34 Définition et proposition (Équivalence de sémantiques dénotationnelles)

Deux sémantiques dénotationnelles sont dites équivalentes *s'il existe un isomorphisme d'$\mathcal{A}$-algèbres ordonnées entre leurs $\mathcal{A}$-algèbres respectives. Dans ce cas, leurs interprétations respectives, notées $[\![-]\!]^1_-$ et $[\![-]\!]^2_-$, vérifient :*

$$\forall\, e, \Gamma\,.\ \theta([\![e]\!]^1_\Gamma) = [\![e]\!]^2_{\theta \circ \Gamma}\,,$$

où θ est l'isomorphisme d'$\mathcal{A}$-algèbres ordonnées.

Démonstration
On note $({D^1}_\perp, \leq, \mathrm{map}, [\![-]\!]^1_-)$ et $({D^2}_\perp, \leq, \mathrm{map}, [\![-]\!]^2_-)$ les deux sémantiques. Rappelons les propriétés d'isomorphisme de θ : c'est une bijection de ${D^1}_\perp$ dans ${D^2}_\perp$, croissante, envoyant donc $\perp$ sur $\perp$, et vérifiant pour toutes dénotations d et d' de D^1, $\theta(\mathrm{map}(d, d')) = \mathrm{map}(\theta(d), \theta(d'))$.
On montre par récurrence sur e que pour tout D^1-environnement Γ, on a

$$\theta([\![e]\!]^1_\Gamma) = [\![e]\!]^2_{\theta\circ\Gamma}\,.$$

- $e = x$

C'est immédiat.

- $e = \lambda x\, a$

Soit $d' \in D^2$, et d son antécédent par θ. On a :

$$\begin{aligned}
\mathrm{map}(\theta([\![\,\lambda x\, a]\!]^1_\Gamma), d') &= \mathrm{map}(\theta([\![\,\lambda x\, a]\!]^1_\Gamma), \theta(d)) \\
&= \theta(\mathrm{map}([\![\,\lambda x\, a]\!]^1_\Gamma, d)) \\
&= \theta([\![a]\!]^1_{\Gamma.(x\,:\,d)}) \\
&= [\![a]\!]^2_{\theta\circ(\Gamma.(x\,:\,d))} \quad \text{(hyp. de réc.)} \\
&= [\![a]\!]^2_{(\theta\circ\Gamma).(x\,:\,d')} \\
&= \mathrm{map}([\![\,\lambda x\, a]\!]^2_{\theta\circ\Gamma}, d')\,.
\end{aligned}$$

Comme d'habitude, on conclut à l'égalité de $\theta([\![\,\lambda x\, a]\!]^1_\Gamma)$ et de $[\![\,\lambda x\, a]\!]^2_{\theta\circ\Gamma}$ par extensionalité ([EXT]), comme les deux dénotations $[\![\,\lambda x\, a]\!]^1_\Gamma$ et $[\![\,\lambda x\, a]\!]^2_{\theta\circ\Gamma}$ appartiennent à D^1 et D^2 respectivement (d'après [VAL]) et comme $\theta^{-1}(\perp) = \{\perp\}$.

- $e = e_1\, e_2$

On a :

$$\begin{aligned}
\theta([\![e_1\, e_2]\!]^1_\Gamma) &= \theta(\mathrm{map}([\![e_1]\!]^1_\Gamma, [\![e_2]\!]^1_\Gamma)) \\
&= \mathrm{map}(\theta([\![e_1]\!]^1_\Gamma), \theta([\![e_2]\!]^1_\Gamma)) \\
&= \mathrm{map}([\![e_1]\!]^2_{\theta\circ\Gamma}, [\![e_2]\!]^2_{\theta\circ\Gamma}) \quad \text{(hyp. de réc.)} \\
&= [\![e_1\, e_2]\!]^2_{\theta\circ\Gamma}\,.
\end{aligned}$$

⌞

On peut finalement déduire de la complète adéquation que les propriétés définissant une sémantique observationnelle la déterminent de manière unique, à un isomorphisme (de $\mathcal{A}$-algèbre ordonnée) près : elles constituent une axiomatisation catégorique de cette sémantique.

2.2.35 Théorème (Unicité de la sémantique observationnelle)
Il existe une unique sémantique observationnelle (suivant la définition 2.2.31 (p. 168)), à un isomorphisme près.

Démonstration
L'existence découle du théorème 2.2.29 (p. 165). Montrons l'unicité, à un isomorphisme près. Supposons deux sémantiques observationnelles et montrons qu'elles sont équivalentes.
On note $(D^1{}_\perp, \leq, \mathrm{map}, [\![-]\!]^1_-)$ et $(D^2{}_\perp, \leq, \mathrm{map}, [\![-]\!]^2_-)$ les deux sémantiques. On quotiente l'ensemble des programmes par la relation d'équivalence contextuelle $=_{\mathrm{C}}$, puis on construit deux bijections strictement croissantes de l'ensemble quotient vers $D^1{}_\perp$ et $D^2{}_\perp$ respectivement, grâce à la propriété de complète adéquation. Si l'on appelle φ et ψ ces bijections, on constate alors que $\psi \circ \varphi^{-1}$ est un isomorphisme de $\mathcal{A}$-algèbres ordonnées.
On note ${\Lambda^0}_{/=_{\mathrm{C}}}$ le quotient de Λ^0 par $=_{\mathrm{C}}$, et si e est un programme, $e_/$ la classe de e modulo $=_{\mathrm{C}}$.
Définissons φ de ${\Lambda^0}_{/=_{\mathrm{C}}}$ dans $D^1{}_\perp$ par $\varphi(e_/) = [\![e]\!]^1_\emptyset$. Évidemment, comme la sémantique est complètement adéquate, $\varphi(e_/)$ ne dépend pas du choix de e.
De même, définissons ψ de $\Lambda^0/=_{\mathrm{C}}$ dans $D^2{}_\perp$ par $\psi(e_/) = [\![e]\!]^2_\emptyset$.
Par les propriétés [DIV] et [ACC], φ et ψ sont surjectives. Par adéquation, φ et ψ sont injectives. Par la complétude de l'adéquation, φ et ψ sont croissantes.
Finalement, soit $\theta = \psi \circ \varphi^{-1}$. C'est une bijection croissante de $D^1{}_\perp$ vers $D^2{}_\perp$. Il reste à montrer que c'est un morphisme de $\mathcal{A}$-algèbres ordonnées.
Soient f et d appartenant à $D^1{}_\perp$, $e_/$ et ${e'}_/$ leurs antécédents par φ. On a :

$$\begin{aligned}
\theta(\mathrm{map}(f,d)) &= \theta(\mathrm{map}([\![e]\!]^1_\emptyset, [\![e']\!]^1_\emptyset)) \\
&= \theta([\![e\,e']\!]^1_\emptyset) \\
&= [\![e\,e']\!]^2_\emptyset \\
&= \mathrm{map}([\![e]\!]^2_\emptyset, [\![e']\!]^2_\emptyset) \\
&= \mathrm{map}(\theta(f), \theta(d))\,.
\end{aligned}$$

θ est bien un morphisme de $\mathcal{A}$-algèbres ordonnées.
⌞

Nous pouvons maintenant nous exercer à prouver quelques règles d'inférence admissibles pour les théories équationnelles et inéquationnelles induites sur les termes par l'équivalence $=_{\mathrm{O}}$ et le pré-ordre $\leq_{\mathrm{O}}$ associés à la sémantique observationnelle[28]. Nous construisons nos raisonnements à partir des axiomes définissant cette sémantique. Voici un jeu minimal de règles

[28] Les théories *équationnelles* et *inéquationnelles* d'une sémantique dénotationnelle cor-

classiques, dont certaines peuvent être montrées pour toute sémantique dénotationnelle. Nous utilisons la notation $(\Lambda, \leq_{\mathrm{O}}, =_{\mathrm{O}})$ pour la réunion des théories équationnelles et inéquationnelles.

2.2.36 Proposition (Structure fine de la sémantique observationnelle) *Considérons la sémantique observationnelle de la définition 2.2.31 (p. 168). Les règles d'inférence suivantes sont admissibles dans la théorie associée* $(\Lambda, \leq_{\mathrm{O}}, =_{\mathrm{O}})$ *:*

– *congruence :*

$$\frac{\emptyset}{x \leq_{\mathrm{O}} x}\ [\text{CONG-VAR}],$$

$$\frac{e \leq_{\mathrm{O}} e'}{\lambda x\, e \leq_{\mathrm{O}} \lambda x\, e'}\ [\text{CONG-ABS}],$$

$$\frac{e_1 \leq_{\mathrm{O}} e_1' \quad e_2 \leq_{\mathrm{O}} e_2'}{e_1\, e_2 \leq_{\mathrm{O}} e_1'\, e_2'}\ [\text{CONG-APP}],$$

– *compatibilité du destructeur d'abstractions :*

$$\frac{\lambda x\, e \leq_{\mathrm{O}} \lambda x\, e'}{e \leq_{\mathrm{O}} e'}\ [\text{DES-ABS}],,$$

– *passage au quotient des substitutions par valeurs :*

$$\frac{e \leq_{\mathrm{O}} e' \quad v \leq_{\mathrm{O}} v'}{e[v/x] \leq_{\mathrm{O}} e'[v'/x]}\ [\text{SUB}] \quad (v, v' \in \boldsymbol{V}),$$

– *stabilité par extension opérationnelle :*

$$\frac{(e[\gamma] \leq_{\mathrm{O}} e'[\gamma])_{\gamma \in \boldsymbol{V}^{\boldsymbol{X}}}}{e \leq_{\mathrm{O}} e'}\ [\text{STA-OP}],$$

– *inclusion du pré-ordre opérationnel :*

$$\frac{\emptyset}{e \leq_{\mathrm{O}} e'}\ [\text{OP}] \quad (e \leq_{\mathrm{op}} e'),$$

respondent exactement aux extensions de l'équivalence et du pré-ordre dénotationnels. Traditionnellement, on les décrit sous la forme d'équations ou d'inéquations, comme nous l'avons fait jusqu'à maintenant. Ces théories forment ce qu'on appelle la *structure fine* de la sémantique.

– *extensionalité opérationnelle :*

$$\frac{(e\,v \leq_{\mathrm{O}} e'\,v)_{v\in \boldsymbol{V}}}{\lambda x\,e\,x \leq_{\mathrm{O}} \lambda x\,e'\,x}\ [\text{EXT-OP}] \quad (x \notin \mathbf{FV}(e) \cup \mathbf{FV}(e'))\,,$$

– *η-conversion :*

$$\frac{\emptyset}{\lambda x\,e\,x =_{\mathrm{O}} e}\ [\text{ETA}] \quad (x \notin \mathbf{FV}(e), \forall\,\gamma\,.\ e[\gamma] \text{ converge})\,.$$

Démonstration

• [CONG] : congruence

C'est le lemme 2.2.24 (p. 161), valable pour toute sémantique dénotationnelle.

• [DES-ABS] : compatibilité du destructeur d'abstractions

Supposons $\lambda x\,e \leq_{\mathrm{O}} \lambda x\,e'$ et considérons un environnement Γ. On a :

$$\begin{array}{rcl} [\![e]\!]_\Gamma & = & \mathrm{map}([\![\,\lambda x\,e]\!]_\Gamma, \Gamma(x)) \\ & = & \mathrm{map}([\![\,\lambda x\,e']\!]_\Gamma, \Gamma(x)) \\ & = & [\![e']\!]_\Gamma\,. \end{array}$$

Finalement, $e \leq_{\mathrm{O}} e'$.

Remarquons que cette règle est valable pour toute sémantique dénotationnelle.

• [SUB] : passage au quotient des substitutions par valeurs

C'est une conséquence immédiate du lemme 2.2.23 (p. 160), valable pour toute sémantique dénotationnelle.

• [STA-OP] : stabilité par extension opérationnelle

C'est le lemme 2.2.32 (p. 170), démontré en utilisant la condition d'accessibilité [ACC]. Cette règle est utilisée dans la suite pour se limiter au cas des programmes, conformément à la description donnée après le lemme 2.2.32.

• [OP] : inclusion du pré-ordre opérationnel

Supposons $e \leq_{\mathrm{op}} e'$, ce qui signifie que pour tout environnement opérationnel γ, ou bien $e[\gamma]$ diverge, ou bien $e[\gamma]$ et $e'[\gamma]$ convergent vers la même valeur.

Comme $\forall\,\gamma \in \mathbf{V}^{\mathbf{X}}\,.\ e[\gamma] \leq_{\mathrm{op}} e'[\gamma]$, par [STA-OP], on peut se limiter au cas où e et e' sont des programmes.

On montre que $e \leq_{\mathrm{O}} e'$, autrement dit $[\![e]\!]_\emptyset \leq [\![e']\!]_\emptyset$.

Supposons que e diverge opérationnellement. Par [DIV], $[\![e]\!]_\emptyset = \bot$, et donc $[\![e]\!]_\emptyset \leq [\![e']\!]_\emptyset$.

Supposons que e converge vers v. Il en est de même de e', et par le lemme

2.2.25 (p. 162), $[\![e]\!]_\emptyset = [\![v]\!]_\emptyset = [\![e']\!]_\emptyset$.
• [EXT-OP] : extensionalité opérationnelle
Comme pour tout environnement opérationnel γ, et tout terme e tel que $x \notin \mathbf{FV}(e)$, on a $(\lambda x\, e\, x)[\gamma] = \lambda x\, e[\gamma]\, x$, alors par [SUB] et [STA-OP], il est suffisant de considérer deux programmes e et e'.
Supposons $\forall\, v \in \mathbf{V}\,.\ e\, v \leq_{\mathrm{O}} e'\, v$.
Considérons un élément d de D, auquel on associe par [ACC] une valeur v telle que $d = [\![v]\!]_\emptyset$.
On a :

$$\begin{array}{rcl} \mathrm{map}([\![\, \lambda x\, e\, x]\!]_\emptyset, d) & = & \mathrm{map}([\![\, \lambda x\, e\, x]\!]_\emptyset, [\![v]\!]_\emptyset) \\ & = & [\![(\lambda x\, e\, x)\, v]\!]_\emptyset \\ & = & [\![e\, v]\!]_\emptyset \quad [\mathrm{OP}] \\ & \leq & [\![e'\, v]\!]_\emptyset \\ & = & [\![(\lambda x\, e'\, x)\, v]\!]_\emptyset \quad [\mathrm{OP}] \\ & = & \mathrm{map}([\![\, \lambda x\, e'\, x]\!]_\emptyset, d)\,. \end{array}$$

Par [VAL] et [EXT], comme à l'habitude, on déduit l'inégalité $[\![\, \lambda x\, e\, x]\!]_\emptyset \leq [\![\, \lambda x\, e'\, x]\!]_\emptyset$.
• [ETA] : η-conversion
Comme pour tout environnement opérationnel γ, on a $(\lambda x\, e\, x)[\gamma] = \lambda x\, e[\gamma]\, x$, dès lors que $x \notin \mathbf{FV}(e)$, par [STA-OP], on peut se limiter au cas où e est un programme.
Soit $d \in D$. On a[29] :

$$\begin{array}{rcl} \mathrm{map}([\![\, \lambda x\, e\, x]\!]_\emptyset, d) & = & [\![e\, x]\!]_{\emptyset.(x\,:\,d)} \\ & = & \mathrm{map}([\![e]\!]_\emptyset, d)\,. \end{array}$$

Comme e est convergent par hypothèse, par le lemme 2.2.25 (p. 162) et [VAL], on a $[\![e]\!]_\emptyset \neq \bot$, et par [EXT], on obtient $[\![\, \lambda x\, e\, x]\!]_\emptyset = [\![e]\!]_\emptyset$.
⌞

Toutes les règles précédentes ont été démontrées par un usage direct des axiomes suivants :
- les axiomes communs à toute sémantique dénotationnelle, [REC] et [VAL],
- celui qui entraîne l'adéquation, [DIV],

[29] Pour que la notation soit cohérente, $\emptyset$ doit représenter ici un environnement quelconque.

– l'axiome d'accessibilité, [ACC], qui permet la correspondance entre les dénotations et les valeurs,
– l'axiome d'extensionalité forte, [EXT+], utilisé uniquement sous sa forme plus faible, [EXT], exprimant l'extensionalité.

Par ailleurs, seules quelques informations élémentaires concernant la sémantique opérationnelle ont été nécessaires : la définition de la relation de réduction et d'évaluation. C'est à rapprocher du fait que la propriété d'extensionalité forte [EXT+] n'a pas été utilisée, mais seulement la propriété d'extensionalité simple [EXT]. Or c'est ce renforcement de l'extensionalité qui est décisif, avec l'accessibilité, dans la démonstration de complète adéquation, permettant de passer de l'équivalence contextuelle à l'équivalence dénotationnelle. Voyons maintenant un exemple de règle où nous utilisons la complète adéquation dans la démonstration : il s'agit ainsi d'utiliser l'équivalence contextuelle pour démontrer l'équivalence dénotationnelle.

Montrons qu'un certain opérateur de point fixe calcule les points fixes par itération.
Définissons le combinateur de point fixe de Curry[30], dans sa version correspondant à l'appel par valeur :

$$Y \stackrel{def}{=} \lambda f\, C(f)\, C(f)\,,$$

où pour tout terme e,

$$C(e) \stackrel{def}{=} \lambda x\, \eta(e\,(x\,x)) \quad (x \notin \mathbf{FV}(e))\,,$$

$\eta(e)$ réalisant l'η-expansion de e, soit

$$\eta(e) \stackrel{def}{=} \lambda x\, e\, x \quad (x \notin \mathbf{FV}(e))\,.$$

Considérons une valeur f et réduisons $Y\, f$:

$$Y\, f \xrightarrow{r} C(f)\, C(f) \xrightarrow{r} \eta(f\,(C(f)\, C(f)))\,.$$

L'η-expansion a ainsi pour rôle d'assurer la terminaison, et est inutile dans le cas de l'appel par nom.
Pour tout terme f, désignons par $\mathrm{fix}(f)$ le terme $C(f)\, C(f)$. Considérons un terme f et un environnement opérationnel γ ; nous avons

$$\mathrm{fix}(f)[\gamma] = \mathrm{fix}(f[\gamma]) \xrightarrow{r} \eta(f[\gamma]\, \mathrm{fix}(f[\gamma])) = \eta(f\, \mathrm{fix}(f))[\gamma]\,.$$

[30]Il traduit le paradoxe de Russell de la théorie (naïve) des ensembles, l'équivalence $\{x \mid x \notin x\} \in \{x \mid x \notin x\} \Leftrightarrow \neg(\{x \mid x \notin x\} \in \{x \mid x \notin x\})$, le paramètre f correspondant à la négation, $x \in y$ se traduisant en l'application $y\, x$ et $\{x \mid \ldots\}$ en $\lambda x \ldots$.

Ainsi les termes $\mathrm{fix}(f)$ et $\eta(f\,\mathrm{fix}(f))$ sont équivalents opérationnellement. C'est la raison qui nous a amené à définir $\mathrm{fix}(f)$ par $C(f)\,C(f)$, et non par $Y\,f$, nous permettant d'éviter de considérer l'évaluation de $Y\,f$, et son éventuelle divergence à cause de f.
La dénotation de $\mathrm{fix}(f)$ est effectivement un point fixe de la dénotation de f, si on suppose que l'application de f à n'importe quelle valeur est convergente, par exemple si on suppose que f a la forme $(\lambda y\,\lambda x\ldots)$. En effet, on a, pour tout environnement Γ :

$$\begin{aligned}[\![\mathrm{fix}(f)]\!]_\Gamma &= [\![\eta(f\,\mathrm{fix}(f))]\!]_\Gamma \quad [\mathrm{OP}]\\ &= [\![f\,\mathrm{fix}(f)]\!]_\Gamma \quad [\mathrm{ETA}]\\ &= \mathrm{map}([\![f]\!]_\Gamma, [\![\mathrm{fix}(f)]\!]_\Gamma)\,.\end{aligned}$$

Nous montrons maintenant que la dénotation de $\mathrm{fix}(f)$ peut être calculée par itération, en partant de la fonction indéfinie en tous ses arguments, puis en itérant par la dénotation de f. Précisément, plutôt que de travailler avec les dénotations, nous préférons les termes, pour établir une nouvelle règle admissible dans la théorie équationnelle induite par l'équivalence dénotationnelle.
Comme nous allons utiliser la propriété d'adéquation complète, nous allons raisonner à partir de l'équivalence contextuelle. Un travail préparatoire va nous être utile, l'étude de la réduction des contextes.

Nous allons utiliser des contextes à une place[31], notée $-$, et étudier leur décomposition en valeurs ou en radicaux dans un environnement de réduction. Conformément à la définition déjà donnée (cf. p. 145), un contexte se définit à partir d'une expression, et non d'un terme, comme l'application qui associe à chaque terme le résultat de sa greffe en chaque occurrence de la place dans l'expression. Cependant, pour l'étude de la décomposition qui nous intéresse, seuls des contextes clos (des programmes à une place) remplissent la place, autrement dit, un contexte peut s'y définir comme une application de l'ensemble des contextes clos dans l'ensemble des contextes. Dans ce cas, la construction d'un contexte à partir d'une expression est compatible avec l'α-conversion, puisque la place d'un contexte clos ne peut pas être liée ; de plus, la correspondance entre les termes à une place et les contextes forme une bijection. Ainsi, pour l'étude de la décomposition des contextes, on considère de manière plus simple qu'un contexte est un terme

[31] Rappelons que la place est une variable réservée à ce seul usage ; en particulier, elle ne peut pas être liée.

à une place.
Nous pouvons alors généraliser des définitions déjà données pour les programmes, contextes particuliers où la place n'apparaît pas.
Un contexte est un terme à une place :

$$C \quad ::= \quad - \mid x \mid \lambda x\, C \mid C\, C\,.$$

Un contexte est clos s'il ne possède aucune variable libre (autre que la place qu'on ne compte pas comme variable libre) :

$$\mathbf{FV}(C) = \emptyset\,.$$

Une valeur est un contexte clos, égal à une abstraction :

$$v \quad ::= \quad \lambda x\, C\,.$$

Un radical est un contexte clos, égal soit à la place, soit à l'application d'une abstraction à une valeur :

$$r \quad ::= \quad - \mid (\lambda x\, C)\, v\,.$$

Un contexte de réduction est un contexte clos à deux places, $-$ et $\underline{\ }^{\mathrm{r}}$, construit autour de la place réservée au radical, $\underline{\ }^{\mathrm{r}}$, en appliquant des valeurs, ou en passant des contextes quelconques :

$$E \quad ::= \quad \underline{\ }^{\mathrm{r}} \mid v\, E \mid E\, C\,.$$

Tout contexte clos peut se décomposer de manière unique soit en une valeur, soit en un radical placé dans un contexte de réduction : si C est un contexte clos, ou bien c'est une valeur, ou bien il existe un unique radical r et un unique contexte de réduction E tel que $C = E[-, r]$ ($E[-, r]$ abrégeant $E[-/-, r/\underline{\ }^{\mathrm{r}}]$).
Nous pouvons énoncer le résultat annoncé, en utilisant la notation développée dans le paragraphe précédent.

2.2.37 Théorème (Point fixe et itération)
Considérons un terme f et définissons la suite des termes itérés par f ainsi :

$$\begin{aligned} f_0 &= \eta(\Omega) \quad (\Omega = (\lambda x\, x\, x)\, (\lambda x\, x\, x))\,, \\ f_{n+1} &= \eta(f\, f_n) \quad (n \in \mathbb{N})\,. \end{aligned}$$

Alors la suite $(f_n)_n$ est croissante suivant $\leq_{\mathrm{O}}$ et admet une borne supérieure vérifiant :

$$\bigvee_{n=0}^{\omega} f_n =_{\mathrm{O}} \mathrm{fix}(f)\,.$$

Démonstration
On doit montrer que

$$\forall\, n\,.\ f_n \leq_{\mathrm{O}} \mathrm{fix}(f)$$

et

$$\forall\, g\,.\ (\forall\, n\,.\ f_n \leq_{\mathrm{O}} g) \Rightarrow \mathrm{fix}(f) \leq_{\mathrm{O}} g\,.$$

Étant donné un environnement opérationnel γ quelconque, il est facile de vérifier que pour tout n, $f_n[\gamma] = (f[\gamma])_n$ (par récurrence) et que $\mathrm{fix}(f)[\gamma] = \mathrm{fix}(f[\gamma])$.
Ainsi, par [SUB] et [STA-OP], on peut se limiter au cas où f et g sont des programmes.
• croissance de $(f_n)_n$
On montre par récurrence sur n que $f_n \leq_{\mathrm{O}} f_{n+1}$.
◦ $n = 0$
Comme $[\![\Omega]\!]_\emptyset = \bot$, on a $\Omega \leq_{\mathrm{O}} f\ f_0$, puis par congruence, $f_0 \leq_{\mathrm{O}} f_1$.
◦ $n > 0$
Supposons $f_{n-1} \leq_{\mathrm{O}} f_n$. Par congruence, on déduit que $f_n \leq_{\mathrm{O}} f_{n+1}$.
• $\mathrm{fix}(f) =_{\mathrm{O}} \eta(f\ \mathrm{fix}(f))$
On a vu précédemment que ces deux programmes sont équivalents opérationnellement. Par application de la règle [OP] de préservation de l'équivalence opérationnelle, ils sont donc équivalents observationnellement.
• $\forall\, n\,.\ f_n \leq_{\mathrm{O}} \mathrm{fix}(f)$
On procède par récurrence sur n.
◦ $n = 0$
Comme $[\![\Omega]\!]_\emptyset = \bot$, on a $\Omega \leq_{\mathrm{O}} f\ \mathrm{fix}(f)$, puis par congruence, $f_0 \leq_{\mathrm{O}} \mathrm{fix}(f)$.
◦ $n > 0$
C'est immédiat, à partir de l'hypothèse de récurrence $f_{n-1} \leq_{\mathrm{O}} \mathrm{fix}(f)$, par congruence et en utilisant l'équivalence $\mathrm{fix}(f) =_{\mathrm{O}} \eta(f\ \mathrm{fix}(f))$.
• $\forall\, g\,.\ (\forall\, n\,.\ f_n \leq_{\mathrm{O}} g) \Rightarrow \mathrm{fix}(f) \leq_{\mathrm{O}} g$
Par complète adéquation, on peut remplacer l'équivalence et le pré-ordre observationnels par l'équivalence et le pré-ordre contextuels.
Soit g un programme vérifiant $\forall\, n\,.\ f_n \leq_{\mathrm{O}} g$. On montre que pour tout contexte clos C, si $C[\mathrm{fix}(f)]$ converge, alors il existe un entier n tel que $C[f_n]$ converge. On en déduira que $C[g]$ converge, puisque $f_n \leq_{\mathrm{C}} g$ par hypothèse, et finalement que $\mathrm{fix}(f) \leq_{\mathrm{C}} g$.
Montrons par récurrence sur l'entier l la propriété suivante : pour tout contexte clos C tel que la trace d'exécution de $C[\mathrm{fix}(f)]$ a pour longueur l, il existe un entier n tel que $C[f_n]$ converge.
Soit C un contexte clos tel que la longueur de la trace d'exécution de $C[\mathrm{fix}(f)]$ est l.

◦ $l = 0$
Nécessairement C est une valeur, et $n = 0$ convient.
◦ $l > 0$
Le contexte C ne peut être une valeur, et donc il peut se décomposer en un radical r dans un contexte de réduction E, soit $C = E[-, r]$. Examinons les deux cas possibles, suivant la nature du radical r.
◦ $r = -$
Dans ce cas, pour tout programme e se réduisant en e', $E[e, e]$ se réduit en $E[e, e']$.
En particulier, $E[\mathrm{fix}(f), \mathrm{fix}(f)]$ se réduit en $E[\mathrm{fix}(f), \eta(f\,\mathrm{fix}(f))]$. Par l'hypothèse de récurrence appliquée au programme $E[-, \eta(f\,-)][\mathrm{fix}(f)]$, il existe n tel que $E[-, \eta(f\,-)][f_n]$ converge.
Comme $E[-, \eta(f\,-)][f_n] = E[f_n, f_{n+1}]$, et comme $f_n \leq_{\mathrm{O}} f_{n+1}$, on déduit par congruence que $E[f_{n+1}, f_{n+1}]$ converge, autrement dit que $C[f_{n+1}]$ converge.
◦ $r = (\lambda x\, D)\, v$
Dans ce cas, pour tout programme e, $E[e, r[e]]$ se réduit en $E[e, D[v/x][e]]$.
En particulier, $E[\mathrm{fix}(f), r[\mathrm{fix}(f)]]$ se réduit en $E[\mathrm{fix}(f), D[v/x][\mathrm{fix}(f)]]$. Par l'hypothèse de récurrence appliquée au programme $E[-, D[v/x]][\mathrm{fix}(f)]$, il existe n tel que $E[-, D[v/x]][f_n]$ converge, et donc tel que $E[f_n, r[f_n]]$ converge, soit $C[f_n]$.
⌞

En conclusion, notre objectif semble rempli de manière satisfaisante. On peut construire une sémantique dénotationnelle du λ-calcul paresseux à partir de la sémantique opérationnelle, en utilisant l'observation des programmes : cette sémantique, dite observationnelle, est complètement adéquate, autrement dit l'équivalence qu'elle induit entre termes égale l'équivalence contextuelle. Cette sémantique peut être axiomatisée simplement, ce qui permet de déduire facilement de nombreuses équivalences entre termes. Outre les axiomes, la démonstration de ces équivalences utilise des propriétés élémentaires de la relation de réduction ; dans les exemples traités, deux méthodes se dessinent : on peut chercher à démontrer ou bien l'équivalence induite par la sémantique observationnelle, ce qui est suffisant dans la plupart des cas, ou bien l'équivalence contextuelle, ce qui demande une analyse plus approfondie de la relation de réduction, précisément dans le seul cas rencontré, l'étude de la réduction des contextes.

Chapitre 3

Deux analyses de l'environnement local

Sommaire

C'est dans ce chapitre que nous exposons notre thèse : il est possible d'analyser l'environnement local seul pour garantir la sécurité d'exécution du code mobile.
Rappelons brièvement notre modèle d'étude : le code mobile s'exécute dans un environnement local, chargé de protéger les ressources appartenant au système hôte. Deux analyses sont présentées, concernant le contrôle respectivement des flux d'informations et des accès. Elles permettent de vérifier que l'environnement local garantit lors de l'exécution de tout code mobile envisageable, d'une part la confidentialité des informations contenues dans certaines ressources locales, d'autre part le confinement des ressources dont

l'accès doit être contrôlé.
De manière plus concrète, nous modélisons le code mobile par un programme, dit mobile, et l'environnement local par un autre programme. Ces deux programmes appartiennent à un langage de haut niveau, modélisé à l'aide du λ-calcul paresseux, simple ou enrichi. L'exécution du code mobile dans l'environnement local est modélisée par la trace d'exécution de l'application du programme mobile à l'environnement local, ce qui suppose que nous munissons le langage d'une sémantique opérationnelle.
Pour faciliter la description de notre méthode, il est utile de formuler les problèmes de flux d'informations et d'accès sous une forme commune : on peut en effet considérer que dans ces deux cas, il s'agit d'observer à la frontière entre le code mobile et le code local un phénomène particulier. Dans le cas des flux d'informations, la frontière est définie de manière globale et statique, comme lieu d'expérimentation pour l'environnement local, auquel on applique différents programmes mobiles ; le phénomène à observer, lui, n'apparaît pas simplement, dans la mesure où il s'agit d'observer le résultat de l'exécution pour déterminer s'il révèle des informations initialement contenues dans les ressources locales. Comment l'information se propage-t-elle lors de l'exécution ? Telle est la première question. Dans le cas des accès, le phénomène à observer se définit simplement comme l'accès à une ressource locale à protéger ; la frontière est difficile à cerner, étant locale et dynamique, puisque c'est le lieu des interactions entre le code mobile et le code local. Le problème est ici de suivre l'évolution de la frontière au cours de l'exécution, à partir de la situation initiale où le code mobile et le code local sont séparés.
Il s'avère qu'une solution générale à ce genre de problèmes existe : l'étiquetage (on dit aussi l'annotation) des termes du langage. Chaque opérateur du langage est annoté par une étiquette qui peut intervenir dans la réduction d'un programme. L'ensemble des étiquettes forme un monoïde, muni en plus de deux opérations binaires, β et σ, dont la justification se comprend à partir de la définition de la β-réduction :

$$@^m(\lambda x^n\, b, a) \to (b[a^{\sigma(m,n)}/x])^{\beta(m,n)}\,.$$

Dans cette réduction, $@^m(-,-)$ est l'opérateur d'application étiqueté par m, $\lambda{-}^n\,-$, l'opérateur d'abstraction étiqueté par n ; de plus, si e est un terme étiqueté égal à $f^j(\ldots)$, alors e^k est égal à $f^{j\,k}(\ldots)$. La fonction β traduit ainsi dans l'étiquette finale de b la β-réduction, alors que la fonction σ traduit dans l'étiquette finale de l'argument a la substitution réalisée. Cette formulation générale s'inspire du langage défini par Lévy dans [50]. Nous la donnons ici non pas parce qu'elle va nous servir sous cette forme

générale par la suite, mais parce que d'une part, il est intéressant de noter qu'elle fournit un outil particulièrement puissant pour l'étude du λ-calcul, comme en témoigne par exemple l'article de Bethke, Klop et de Vrijer [16], d'autre part elle met en évidence les différences de traitements entre les flux d'informations et les accès. En effet, pour l'étude de la confidentialité et du confinement, nous utilisons deux langages étiquetés distincts, qui peuvent être considérés comme deux cas extrêmement simples d'annotation.
Pour la confidentialité, on suppose qu'une étiquette est soit le mot vide, utilisé pour le code public, soit δ, utilisé pour le code confidentiel, ou privé ; pour la concaténation, on suppose que δ est un élément absorbant ; quant aux fonctions β et σ, elles sont définies par $\beta(m,n) = n\,m$ et $\sigma(m,n) = \varepsilon$ (le mot vide), ce qui donne la β-réduction suivante :

$$@^m(\lambda x^n\, b, a) \quad \to \quad (b[a/x])^{n\,m}\,.$$

La définition de β permet de propager les dépendances, en affirmant que le résultat dépend de l'abstraction et de l'application. Noter que ce langage étiqueté correspond à l'appel par nom, mais non à l'appel par valeur que nous utilisons, et où il faudrait tenir compte de l'étiquette de l'argument, qui peut diverger avant de remplacer la variable x ; c'est uniquement pour simplifier cette présentation rapide que nous avons fait ce choix. Adapté à l'étude des flux d'informations, ce langage étiqueté permet de déterminer si le résultat observable d'un programme paramétré dépend de ses paramètres. Représentons un programme paramétré (à un paramètre pour simplifier) par un contexte $C_{[\,]}$ à une place ; le résultat observable du programme paramétré $C_{[\,]}$ dépend du paramètre s'il existe deux termes e_1 et e_2 tels que $C[e_1]$ converge et $C[e_2]$ diverge, puisqu'on n'observe du résultat de l'exécution que la convergence. De deux choses, l'une : ou bien, pour tout terme e, le programme $C[e]$ diverge, et dans ce cas, le résultat observable de $C_{[\,]}$ ne dépend pas de son paramètre, ou bien il existe un terme e tel que $C[e]$ converge. Dans ce dernier cas, considérons l'exécution du programme étiqueté $C[e^\delta]$, où e a été étiqueté par δ ; alors l'étiquette du résultat est différente de δ si et seulement si pour tout terme a, $C[a]$ converge.
L'utilisation de ce langage étiqueté dans ce but n'est pas nouvelle. Dans [28], Gandhe, Venkatesh et Sanyal s'en servent dans le cas du λ-calcul pur (sans stratégie d'évaluation) pour montrer la même propriété ; Klop et al. dans [16, § 7], Abadi, Lampson et Lévy un peu plus tôt dans [3] montrent une autre propriété classique de dépendance, dite de stabilité ou de généricité, où c'est la valeur même du résultat qui est observée ; enfin, Conchon et Pottier dans [70] appliquent les résultats d'Abadi et al. à la question du contrôle des flux d'informations, ouvrant ainsi la voie que nous empruntons.

Intéressons-nous maintenant au confinement. Une étiquette cette fois représente l'origine de l'opérateur, soit le code mobile, soit le code local. Il n'est pas nécessaire de définir la concaténation, puisqu'aucune opération n'est réalisée sur les étiquettes. En effet, $\beta(m,n)$ et $\sigma(m,n)$ sont égaux tous deux au mot vide, ce qui donne pour la réduction :

$$@^m(\lambda x^n\, b, a) \quad \to \quad b[a/x]\,.$$

L'origine des opérateurs est ainsi préservée par réduction. Il est alors possible de définir formellement la frontière entre le code mobile et le code local lors de l'exécution. Un terme e^m est à la frontière dans un programme a s'il existe un sous-terme $f^n(\ldots, e^m, \ldots)$ de a tel que m est distinct de n.
Ce langage étiqueté est utilisé par Klop et al. dans [16, §4] pour définir formellement la notion d'ancêtre d'un opérateur. D'un point de vue sécuritaire, on peut considérer que c'est le modèle sous-jacent des langages permettant d'associer à des ressources des droits d'accès. Lorsqu'un opérateur accède à une ressource, il doit être vérifié que l'opérateur en a le droit, en contrôlant son origine. Les qualificatifs d'accès « public » et « private » attribués aux champs et aux méthodes en Java constituent un exemple de droits associés à un objet, entendu comme ressource. Skalka et Smith dans [78] proposent un raffinement de cette possibilité de qualifier les accès ; la sémantique opérationnelle de leur langage orienté objets se définit à partir d'une relation de réduction qui préserve l'origine du code et les droits d'accès associés aux ressources. Un autre exemple intéressant est le « SLam Calculus »[1] (cf. [34]). Ce langage conçu par Heintze et Riecke permet d'attribuer à chaque ressource une première étiquette liée à l'information qu'elle contient et une seconde définissant les droits d'accès à la ressource, et à chaque opérateur d'accès une origine. La relation de réduction définissant la sémantique opérationnelle réalise la synthèse des deux relations que nous venons de décrire, en propageant les informations et en préservant les origines et les droits d'accès.

Détaillons maintenant le contenu du chapitre. Dans la première partie, l'étude des flux d'informations est menée en deux temps. Tout d'abord, nous étudions la question de la dépendance d'un programme paramétré relativement à la valeur de ses paramètres, puis nous définissons un critère de confidentialité fondé sur une analyse de l'environnement local. Le langage utilisé est le λ-calcul paresseux avec appel par valeur.
À un langage étiqueté comme celui décrit précédemment, nous préférons un langage à deux niveaux, l'un correspondant au code public, l'autre au code privé. C'est une formulation équivalente où les étiquettes annotent des

[1] C'est l'acronyme de « Secure λ-Calculus ».

termes et non des opérateurs[2]. Elle est particulièrement adaptée à notre cas dans la mesure où le code public et le code privé ne s'imbriquent pas : le programme paramétré est en effet formé de code public, alors que les termes remplaçant les paramètres sont entièrement formés de code privé. Elle permet aussi de distinguer nettement deux niveaux non seulement dans la syntaxe, mais aussi dans la sémantique : l'exécution du code public sera modélisée par une relation de réduction, alors que l'exécution du code privé sera modélisée par une relation d'évaluation. C'est cette distinction qui facilite l'interprétation abstraite du code privé.

Pour l'étude de la dépendance, notre démarche se résume ainsi :

- on commence par définir la dépendance : un programme paramétré dépend de ses paramètres si le résultat observable varie lorsque ses paramètres varient ; rappelons qu'on observe la convergence ou la divergence ;
- le langage à deux niveaux permet de traduire un programme paramétré en distinguant le code public appartenant au programme du code privé appartenant aux paramètres ; la propriété de dépendance s'exprime tout aussi bien dans ce langage à deux niveaux ;
- le langage à deux niveaux est interprété abstraitement, en oubliant le code privé, interprété par une unique valeur ; nous montrons que cette traduction abstraite est complètement adéquate pour la dépendance, autrement dit, il est possible de définir abstraitement la dépendance, sans aucune approximation ; ce résultat ne devrait pas surprendre dans la mesure où il est voisin de résultats bien connus, comme le théorème de stabilité cité plus haut.

Pour l'étude de la confidentialité, on considère que l'environnement local est un programme paramétré, noté $\mathsf{L}_{[\ ]}$, le paramètre représentant la ressource dont le contenu doit rester confidentiel. Nous avons vu en introduction (cf. p. 12) que la propriété de confidentialité peut s'exprimer ainsi :

> pour tout couple de valeurs initiales δ_1 et δ_2 du paramètre local, l'environnement local $\mathsf{L}[\delta_1]$ est équivalent contextuellement à l'environnement local $\mathsf{L}[\delta_2]$.

En utilisant la sémantique observationnelle du second chapitre (cf. 2.2.29 (p. 165)), on obtient une propriété équivalente :

> pour tout couple de valeurs initiales δ_1 et δ_2 du paramètre local,

[2]Si on considère une paire d'étiquettes $\{m, n\}$, on définit la traduction par deux fonctions, T_m et T_n, vérifiant $T_m(f^m(e_j)_j) = f(T_m(e_j))_j$ et $T_m(f^n(e_j)_j) = n(f(T_n(e_j))_j)$, et de même pour T_n ; les étiquettes deviennent ainsi des opérateurs unaires. La traduction du terme étiqueté e s'obtient par $T_m(e)$ ou $T_n(e)$.

> l'observation de $\mathsf{L}[\delta_1]$ est égale à l'observation de $\mathsf{L}[\delta_2]$.

Compte tenu de la définition de l'observation (cf. déf. 2.2.1 (p. 139)), elle se récrit encore :

> pour tout couple de valeurs initiales δ_1 et δ_2 du paramètre local,
> pour toute suite de valeurs $v_1, \ldots, v_n$, le programme $\mathsf{L}[\delta_1]\, v_1 \ldots v_n$
> converge si et seulement si le programme $\mathsf{L}[\delta_2]\, v_1 \ldots v_n$ converge,

soit, par définition de la dépendance,

> pour toute suite de valeurs $v_1, \ldots, v_n$, le programme local paramétré $\mathsf{L}_{[\ ]}\, v_1 \ldots v_n$ ne dépend pas de son paramètre.

Comme nous avons pu définir abstraitement la dépendance, il est possible de définir l'observation abstraite de l'environnement local, comme a été définie l'observation concrète, pour arriver finalement au résultat suivant :

> l'environnement local garantit la confidentialité de la ressource
> si et seulement si son observation abstraite ne fait pas apparaître
> de dépendance.

À notre connaissance, ce critère de confidentialité est nouveau. Il est intéressant parce qu'il permet d'adapter les analyses statiques de flux d'informations au cas du code mobile. Généralement, ces analyses sont des variations de celle que Denning décrit dans [26]. Elles reposent donc sur l'idée d'abstraire le code confidentiel par une unique valeur, comme nous venons de le faire. On peut considérer qu'une telle analyse statique associe à chaque programme paramétré une valeur abstraite d'un domaine abstrait, et qu'il est possible de déterminer si une valeur abstraite révèle de l'information confidentielle, autrement dit, révèle une dépendance relativement aux paramètres. L'observation abstraite peut alors être construite à partir de ces valeurs abstraites. La question qui se pose est finalement d'approcher cette observation abstraite en utilisant une représentation finie.

Dans la seconde partie du chapitre, nous nous intéressons à la question du confinement. Il s'agit de vérifier que les ressources à protéger sont confinées dans l'environnement local. C'est une propriété indispensable pour le contrôle des accès, qui complète le contrôle de l'usage des ressources dans le code local, et signifie que le code mobile ne peut pas accéder directement, sans contrôle, aux ressources.

Nous utilisons un langage enrichi par des références, permettant de manipuler des objets en mémoire ; il devient aussi typé, afin de distinguer les objets en mémoire des fonctions. Grâce aux références, il est possible de définir pour chaque type un combinateur de point fixe[3], ce qui est impossible dans

[3] On procède ainsi pour un type fonctionnel $C = A \to B$. Le combinateur de point fixe

le λ-calcul typé, où tous les programmes terminent. Le langage reste donc suffisamment expressif. La sémantique opérationnelle du langage est définie par une relation de réduction entre configurations, formées chacune d'une description de l'état de la mémoire et d'un terme de contrôle.
Comme annoncé, ce langage est annoté de manière à pouvoir retrouver l'origine de chaque opérateur, information indispensable pour vérifier le confinement. Il est alors possible de définir la frontière d'interaction entre le code mobile et le code local lors de l'exécution : un terme apparaît à la frontière dans une configuration s'il est l'argument d'un opérateur d'origine différente, ou le contenu d'une référence d'origine différente. Notre première tâche est de caractériser par leur type les termes apparaissant à la frontière lors d'une exécution. Précisément, à partir de l'ensemble des types apparaissant à la frontière dans une configuration initiale, nous montrons comment calculer un ensemble de types frontaliers qui contient l'ensemble des types apparaissant à la frontière dans toute configuration ultérieure. Ce résultat s'applique alors naturellement au problème du confinement pour le code mobile. Comme nous modélisons l'exécution du code mobile dans son environnement par l'application du programme mobile, fonction à un argument, à l'environnement local, il vient qu'initialement seul l'environnement local est à la frontière, si bien qu'il est possible d'obtenir de l'ensemble des types apparaissant à la frontière une majoration calculée à partir du type de l'environnement local.
Définissons maintenant un type confiné : un type est confiné dans l'environnement local si le code mobile n'opère pas directement sur une valeur de ce type provenant du code local. Pour qu'un type soit confiné, il suffit qu'il ne puisse apparaître à la frontière, ce qui donne en utilisant la majoration obtenue le critère suivant :

> si un type n'apparaît dans le type de l'environnement ni en une occurrence positive, ni sous la garde du constructeur des types de références, alors il est confiné dans l'environnement local.

Ce résultat, nouveau, résout une question posée par Leroy et Rouaix dans [49, §5.1, p. 397]. Il est optimal, puisqu'il donne la condition la plus faible possible à partir des types. En effet, nous montrons que si un type A apparaît dans un type donné L en une occurrence positive, ou sous la garde du

---

est défini sous la forme du contenu d'une référence :

$$\begin{array}{l}\text{let } \mathrm{Ref}((C \to C) \to C)\, Y = a_{\mathrm{Ref}((C\to C)\to C)} \text{ in} \\ \quad \mathrm{set}(Y, \lambda f : C \to C.\, \eta(f\,(\mathrm{get}(Y)\, f)))\,;\, \mathrm{get}(Y)\,,\end{array}$$

où a_A est une valeur par défaut de type A, $\mathrm{set}(l, e)$ affecte e à l, $\mathrm{get}(l)$ lit le contenu de l, let $A\, x = a$ in e déclare et initialise x à a de type A dans e et enfin $\eta(f)$ réalise l'η-expansion de f.

constructeur des types de références, alors il existe un environnement local de type L et un programme mobile tels que le code mobile peut opérer sur une valeur de type A provenant du code local.
Donnons quelques exemples de telles attaques ou coopérations.
Si $L = (A \to B) \to C$, le code mobile passe à l'environnement local une fonction réalisant l'accès, à laquelle l'environnement transmet la valeur de type A.
Si $L = \mathrm{Ref}(A) \to B$, le code mobile transmet une référence, à laquelle l'environnement affecte une valeur de type A, qui devient disponible pour le code mobile.
Si $L = \mathrm{Ref}(A \to B)$, l'exercice est plus difficile. L'environnement affecte à la référence qu'il rend accessible, notée l, une fonction particulière, notée f : celle-ci commence par lire le contenu de l, qui est une fonction, puis lui passe une valeur de type A. Tant que le contenu de l est f, cette fonction ne termine donc en aucun argument. Quant au code mobile, il lit le contenu de l, obtient f, affecte à l une fonction réalisant l'accès à l'argument de type A, et applique finalement f à un argument quelconque : le contenu de l, soit maintenant la fonction d'accès provenant du code mobile, reçoit une valeur de type A d'origine le code local.
Revenons maintenant à la question du contrôle des accès. Comme nous l'avons vu en introduction, les accès aux ressources peuvent être contrôlés d'une part en encapsulant dans les ressources les fonctions de contrôles, d'autre part

- en confinant au code local les ressources à protéger
- et en instrumentant le code local, de manière à contrôler l'usage des ressources à protéger.

C'est évidemment cette seconde voie qui nous intéresse. Dans un premier temps, nous cherchons à vérifier qu'étant donné un programme instrumenté, l'instrumentation du code est correctement effectuée. Nous posons initialement cette question de manière générale, sans nous préoccuper de la distinction entre le code local et le code mobile, si bien qu'il s'agit de déterminer si l'exécution d'un programme vérifie la propriété suivante :

> le code non instrumenté n'accède jamais à une ressource devant être protégée.

Nous ne nous intéressons pas à la manière dont le code est instrumenté, ni par conséquent à la nature de la politique de sécurité que l'instrumentation applique. Comme les ressources sont modélisées dans notre langage par des valeurs, et les fonctions d'accès par des destructeurs, la propriété précédente peut se formuler plus précisément ainsi :

> une valeur devant être protégée n'est jamais détruite par un destructeur non instrumenté.

Syntaxiquement, il est donc nécessaire de pouvoir distinguer les opérateurs à protéger ou instrumentés de ceux qui ne le sont pas. Aussi introduisons-nous deux étiquettes, $\bot$ et $\top$, auxquelles nous attribuons la signification suivante, selon qu'elles annotent un constructeur ou un destructeur :

	constructeur	destructeur
$\bot$	pas de protection	pas d'instrumentation
$\top$	protection	instrumentation

La propriété s'énonce finalement ainsi :

> un destructeur étiqueté par $\bot$ n'est jamais appliqué à une valeur étiquetée par $\top$.

C'est par un système de types annotés que nous imposons le respect de cette propriété : tout programme admettant un type dans ce système vérifie alors la propriété lors de son exécution. Dans ce système, une valeur à protéger reçoit un type étiqueté par $\top$, alors qu'une valeur ne nécessitant aucune protection reçoit un type étiqueté par $\bot$. De tels types étiquetés seront dits sécuritaires. Ce système est précisément la restriction de celui du « SLam Calculus » aux contrôles d'accès. Dans [34], Heintze et Riecke définissent également une relation de sous-typage, où tout type $A^{\bot}$ est un sous-type de $A^{\top}$. C'est qu'en effet le principe de substitution tel qu'il est énoncé par Liskov et Wing dans [51, §1] s'applique. Considérons en effet un programme paramétré quelconque $C_{[\,]}$: si pour tout valeur v de type $A^{\top}$, la propriété précédente est vérifiée lors de l'exécution de $C[v]$, alors on peut en déduire que pour toute valeur v de type $A^{\bot}$, elle l'est également.
Cette relation de sous-typage permet de reformuler la question du confinement en utilisant le slogan « convertir pour confiner ». Le programme mobile est supposé typé dans le système des types classiques, sans annotation, alors que l'environnement local est typé dans le système des types sécuritaires ; cette dissymétrie traduit la distinction entre le code mobile, indéfini, et le code local, à disposition pour l'analyse. Remarquons alors que le programme mobile est aussi typé dans le système des types sécuritaires, à condition d'utiliser la seule étiquette $\bot$, puisqu'il ne contient pas de ressources à protéger et n'est pas instrumenté. En conséquence, si le type sécuritaire de l'environnement local n'utilise que l'étiquette $\bot$, l'application du programme mobile à l'environnement local admet aussi un type dans le système des types sécuritaires, et la propriété de confinement est vérifiée. Si le type de l'environnement local comporte des étiquettes $\top$, il est possible

dans certains cas de les faire disparaître en utilisant les règles de sous-typage. Par exemple, si l'environnement local a pour type $A^\top$, il est impossible de supprimer l'étiquette $\top$, alors que s'il a pour type $A^\top \xrightarrow{\perp} B^\perp$, il a aussi pour type le sur-type $A^\perp \xrightarrow{\perp} B^\perp$, annoté seulement par $\perp$ (en utilisant la contravariance pour l'ensemble de départ). La connexion avec notre étude précédente de la frontière et du confinement vient de la propriété suivante de la relation de sous-typage :

> le type sécuritaire de l'environnement local peut être converti en un type sécuritaire utilisant seulement l'étiquette $\perp$ si et seulement si tout type apparaissant dans le type de l'environnement local en une occurrence positive ou sous la garde du constructeur des types de références est étiqueté par $\perp$.

On retrouve que le contrôle des accès repose d'une part sur l'instrumentation du code local, d'autre part sur une forme de confinement, puisque la dernière propriété implique que les types à protéger sont confinés. C'est cette connexion qui est intéressante, puisqu'elle formalise l'intuition qu'un type définit une frontière.

Terminologie Par la suite, nous allons souvent rencontrer des traductions entre deux ensembles, par exemple une interprétation abstraite permettant de passer d'un ensemble concret à un ensemble abstrait. Ces ensembles en correspondance ne viennent pas seuls, mais munis de fonctions ou de relations, et nous nous intéressons à leur transfert d'un ensemble à l'autre par la traduction. Dans un contexte algébrique, la notion de morphisme est introduite par exemple pour exprimer une compatibilité entre la traduction et les opérations. Voici une description rapide de notre terminologie pour ce genre de correspondances, dans le cas d'une propriété.
On considère deux ensembles disjoints E et F, et une propriété P, identifiée par son extension, définie sur la réunion de E et F. On note P_E l'intersection de P et E, P_F l'intersection de P et F. Soit φ l'application de E dans F réalisant la traduction. On dira que :

- φ *préserve* P, ou P est *stable* par φ, si $\varphi(P_E) \subseteq P_F$,
- P est *dense* pour φ si $P_F \subseteq \varphi(P_E)$,
- φ *conserve* P si $\varphi(P_E) = P_F$,
- φ est *adéquate* pour P si $\varphi^{-1}(P_F) \subseteq P_E$,
- φ est *complètement adéquate* pour P si $\varphi^{-1}(P_F) = P_E$.

Les trois premières définitions suivent la terminologie utilisée pour les points fixes (pré et post). Les deux dernières suivent la terminologie utilisée en logique pour définir le rapport entre un calcul syntaxique (pour démontrer)

et la sémantique (pour signifier) : on calcule dans F, on signifie dans E, ce qui est typiquement le cas lorsqu'on φ correspond à une interprétation abstraite du domaine concret E vers le domaine abstrait F.

3.1 Un critère abstrait de confidentialité

Nous nous intéressons aux flux d'informations entre les ressources de l'hôte et le code mobile. Précisément, nous cherchons à caractériser les environnements garantissant la confidentialité des ressources qu'ils protègent. Notre point de départ est plus général, puisque nous commençons par étudier la question de la dépendance : il s'agit de caractériser les programmes paramétrés dont le résultat observable dépend de leurs paramètres. À cette fin, nous introduisons un langage à deux niveaux, l'un public, correspondant au code du programme paramétré, l'autre privé, correspondant au code des termes remplaçant les paramètres. C'est son interprétation abstraite qui va nous servir à caractériser abstraitement tout d'abord la dépendance, puis la confidentialité. L'interprétation abstraite consiste à interpréter le niveau privé trivialement, par une seule valeur. Rappelons enfin que cette étude est menée à partir du λ-calcul paresseux, avec appel par valeur.

3.1.1 Un langage à deux niveaux

Ajoutons au λ-calcul un second niveau : le premier niveau correspond au code public, le second au code privé, au sens où sa confidentialité doit être assurée. La syntaxe du langage est enrichie par un opérateur permettant de transformer du code privé en code public ; plus précisément, pour faciliter l'expression de la sémantique opérationnelle, deux versions de cet opérateur sont utilisées, la première, notée δ_{r}, permettant de convertir n'importe quel terme privé en un terme public, qui, une fois clos, donne un radical, la seconde, notée δ_{v}, permettant de convertir n'importe quelle abstraction privée en un terme public, qui, une fois clos, donne une valeur.

La syntaxe du λ-calcul à deux niveaux est définie par la grammaire suivante :

$$e \quad ::= \quad x \mid \lambda x\, e \mid e\, e \mid \delta_{\mathrm{r}}(e) \mid \delta_{\mathrm{v}}(\lambda x\, e)\,.$$

Il est important de remarquer que seuls les opérateurs de conversion les plus extérieurs importent : ils marquent la limite entre le code public et le code privé. On note Λ_2 l'ensemble des termes à deux niveaux, et Λ_2^0 l'ensemble des programmes à deux niveaux.

La sémantique opérationnelle est définie en distinguant deux niveaux, par une relation de réduction pour le code public, par une relation d'évaluation pour le code privé ; on la formule cependant en donnant directement le résultat de l'exécution, comme on l'a vu dans la table 2.11 (p. 136) pour le λ-calcul simple. Quelques notions usuelles sont indispensables.
On ajoute à la définition habituelle d'une substitution qu'elle commute avec δ_r et δ_v : si σ est une substitution, on a ainsi

$$\begin{aligned} \delta_\mathrm{r}(e)[\sigma] &= \delta_\mathrm{r}(e[\sigma])\,, \\ \delta_\mathrm{v}(\lambda x\, e)[\sigma] &= \delta_\mathrm{v}((\lambda x\, e)[\sigma])\,. \end{aligned}$$

Une *valeur* est ou bien une abstraction close publique, ou bien la conversion publique d'une abstraction close privée :

$$v \quad ::= \quad \lambda x\, e \mid \delta_\mathrm{v}(\lambda x\, e)\,,$$

où $\lambda x\, e$ est une abstraction close. On note $\mathbf{V}_2$ l'ensemble des valeurs.
Un *radical* est ou bien l'application d'une valeur à une autre valeur, ou bien la conversion publique d'un programme privé :

$$r \quad ::= \quad v\, v \mid \delta_\mathrm{r}(e)\,,$$

où e est un programme.
Un *contexte de réduction* se construit autour de la place, en appliquant des valeurs, ou en passant des arguments :

$$E \quad ::= \quad - \mid v\, E \mid E\, e\,,$$

où e est un programme.
Comme d'habitude, tout programme peut se décomposer de manière unique en une valeur ou un radical placé dans un contexte de réduction : si e est un programme, ou bien c'est une valeur, ou bien il existe un unique radical r et un unique contexte de réduction E tel que $e = E[r]$ ($E[r]$ abrégeant $E[r/-]$).
Enfin, un *résultat d'évaluation* est soit une valeur, soit la divergence ; on distingue deux valeurs de divergence, une par niveau, notées $\bot$ pour le code public, $\delta_\mathrm{v}(\bot)$ pour le code privé, entendue comme la conversion publique de la divergence du code privé :

$$\rho \quad ::= \quad v \mid \bot \mid \delta_\mathrm{v}(\bot)\,.$$

Il est utile de définir la traduction du λ-calcul à deux niveaux permettant d'effacer les conversions et donc d'abolir les niveaux. Cette application, notée $\mathbf{T}_{2\to1}$, est définie récursivement par les équations suivantes :

$$\begin{aligned}
\mathbf{T}_{2\to1}(x) &= x\,,\\
\mathbf{T}_{2\to1}(\lambda x\, e) &= \lambda x\, \mathbf{T}_{2\to1}(e)\,,\\
\mathbf{T}_{2\to1}(f\, e) &= \mathbf{T}_{2\to1}(f)\, \mathbf{T}_{2\to1}(e)\,,\\
\mathbf{T}_{2\to1}(\delta_{\mathrm{r}}(e)) &= \mathbf{T}_{2\to1}(e)\,,\\
\mathbf{T}_{2\to1}(\delta_{\mathrm{v}}(\lambda x\, e))) &= \mathbf{T}_{2\to1}(\lambda x\, e)\,.
\end{aligned}$$

La fonction de traduction $\mathbf{T}_{2\to1}$ est étendue aux résultats d'évaluation par $\mathbf{T}_{2\to1}(\bot) = \bot$ et $\mathbf{T}_{2\to1}(\delta_{\mathrm{v}}(\bot)) = \bot$. On remarque que la fonction de traduction $\mathbf{T}_{2\to1}$ commute avec les substitutions, puisque pour tous termes e_1 et e_2 et toute variable x, on a :

$$\mathbf{T}_{2\to1}(e_1[e_2/x]) = \mathbf{T}_{2\to1}(e_1)[\mathbf{T}_{2\to1}(e_2)/x]\,.$$

Elle traduit aussi une valeur en une valeur, un contexte de réduction en un contexte de réduction et un radical différent de $\delta_{\mathrm{r}}(\lambda x\, e)$ en un radical. Par la suite, on utilisera ces propriétés sans mention.
La relation d'évaluation, notée $\Downarrow_2$, qui associe à un programme un résultat d'évaluation, est définie par le système d'inférence de la table 3.1 (p. 196), pour lequel une interprétation mixte est utilisée ; elle s'appuie sur la relation d'évaluation du λ-calcul simple, toujours notée $\Downarrow$. Les règles se répartissent en trois groupes :

- les règles [VAL], [β+] et [$\beta-$] correspondent à celles du λ-calcul simple, étudié précédemment,
- les règles [PRIV/β+] et [PRIV/$\beta-$] correspondent à l'application à un argument d'une abstraction privée, application publique donnant un résultat privé,
- les règles [PRIV/CONV+], [PRIV/CONV–] et [PRIV/DIV] correspondent à l'exécution du code privé.

Quelques résultats concernant la sémantique opérationnelle nous seront utiles. On montre tout d'abord que la relation d'évaluation se préserve par effacement des niveaux, puis que la relation d'évaluation est déterministe et totale.

3.1.1 Proposition (Lemme de l'effacement)
Soit e un programme à deux niveaux s'évaluant en ρ :

$$e \Downarrow_2 \rho\,.$$

$$\frac{\emptyset}{v \Downarrow_2 v}\,[\text{VAL}] \quad (v \in \mathbf{V}_2)$$

$$\frac{+\,E[a[v/x]] \Downarrow_2 \rho}{E[(\lambda x\, a)\, v] \Downarrow_2 \rho}\,[\beta+] \quad \left(\begin{array}{l}\rho \neq \bot \\ v \in \mathbf{V}_2\end{array}\right)$$

$$\frac{-\,E[a[v/x]] \Downarrow_2 \bot}{E[(\lambda x\, a)\, v] \Downarrow_2 \bot}\,[\beta-] \quad (v \in \mathbf{V}_2)$$

$$\frac{+\,E[\delta_{\text{r}}(a[v/x])] \Downarrow_2 \rho}{E[\delta_{\text{v}}(\lambda x\, a)\, v] \Downarrow_2 \rho}\,[\text{PRIV}/\beta+] \quad \left(\begin{array}{l}\rho \neq \bot \\ v \in \mathbf{V}_2\end{array}\right)$$

$$\frac{-\,E[\delta_{\text{r}}(a[v/x])] \Downarrow_2 \bot}{E[\delta_{\text{v}}(\lambda x\, a)\, v] \Downarrow_2 \bot}\,[\text{PRIV}/\beta-] \quad (v \in \mathbf{V}_2)$$

$$\frac{+\,E[\delta_{\text{v}}(\lambda x\, a)] \Downarrow_2 \rho}{E[\delta_{\text{r}}(e)] \Downarrow_2 \rho}\,[\text{PRIV/CONV}+] \quad \left(\begin{array}{l}\rho \neq \bot \\ \mathbf{T}_{2\to 1}(e) \Downarrow \lambda x\, a\end{array}\right)$$

$$\frac{-\,E[\delta_{\text{v}}(\lambda x\, a)] \Downarrow_2 \bot}{E[\delta_{\text{r}}(e)] \Downarrow_2 \bot}\,[\text{PRIV/CONV}-] \quad (\mathbf{T}_{2\to 1}(e) \Downarrow \lambda x\, a)$$

$$\frac{\emptyset}{E[\delta_{\text{r}}(e)] \Downarrow_2 \delta_{\text{v}}(\bot)}\,[\text{PRIV/DIV}] \quad (\mathbf{T}_{2\to 1}(e) \Downarrow \bot)$$

Tab. 3.1 – Sémantique opérationnelle du langage à deux niveaux

Alors sa traduction dans le λ-calcul simple $\mathbf{T}_{2\to1}(e)$ *s'évalue en la traduction de* ρ, $\mathbf{T}_{2\to1}(\rho)$ *:*

$$\mathbf{T}_{2\to1}(e) \Downarrow \mathbf{T}_{2\to1}(\rho)\,.$$

Démonstration
Pour tout couple (e,ρ) tel que $e \Downarrow_2 \rho$, on construit une preuve de

$$\mathbf{T}_{2\to1}(e) \Downarrow \mathbf{T}_{2\to1}(\rho)$$

en résolvant un système d'équations récursives, d'inconnues $(X_{(e,\rho)})_{(e,\rho)|e\Downarrow_2\rho}$. Pour faciliter une description générique des équations, on note ρ_+ un résultat d'exécution égal à une valeur, et ρ_- une valeur de divergence, $\bot$ ou $\delta_{\mathrm{v}}(\bot)$. L'indice « + » ou « − » se réfère à l'interprétation mixte du système et indique la contrainte portant sur le profil de la preuve de la prémisse, « + » pour une preuve bien fondée, « − » pour une preuve infinie en profondeur. On décrit dans la table 3.2 (p. 197) l'équation associée à $X_{(e,\rho_*)}$ en fonction de la décomposition de e soit en une valeur, soit en un radical dans un contexte de réduction ; on a noté dans cette table $*$ un élément de la paire $\{+,-\}$. À cause de l'équation 3.1, le système n'est pas gardé ; montrons qu'il

$$X_{(v,v)} = \frac{\emptyset}{\mathbf{T}_{2\to1}(v) \Downarrow \mathbf{T}_{2\to1}(v)} \quad (v \in \mathbf{V}_2)$$

$$X_{(E[(\lambda x\, a)\, v],\rho_*)} = \frac{* \; X_{(E[a[v/x]],\rho_*)}}{\mathbf{T}_{2\to1}(E[(\lambda x\, a)\, v]) \Downarrow \mathbf{T}_{2\to1}(\rho_*)}$$

$$X_{(E[\delta_{\mathrm{v}}(\lambda x\, a)\, v],\rho_*)} = \frac{* \; X_{(E[\delta_{\mathrm{r}}(a[v/x])],\rho_*)}}{\mathbf{T}_{2\to1}(E[\delta_{\mathrm{v}}(\lambda x\, a)\, v]) \Downarrow \mathbf{T}_{2\to1}(\rho_*)}$$

$$X_{(E[\delta_{\mathrm{r}}(a)],\rho_*)} = \begin{cases} X_{(E[\delta_{\mathrm{v}}(\mathbf{T}_{2\to1}(a))],\rho_*)} & \text{si } \mathbf{T}_{2\to1}(a) \in \mathbf{V} \\ \dfrac{* \; X_{(E[\delta_{\mathrm{r}}(a')],\rho_*)}}{\mathbf{T}_{2\to1}(E[\delta_{\mathrm{r}}(a)]) \Downarrow \mathbf{T}_{2\to1}(\rho_*)} & \text{sinon} \\ & (\mathbf{T}_{2\to1}(a) \to a') \end{cases} \tag{3.1}$$

Tab. 3.2 – Dém. prop. 3.1.1

est cependant équivalent à un système gardé.
Dans ce but, définissons une relation entre programmes à deux niveaux, notée $\overset{rv}{\to}$, par :

$$e \overset{rv}{\to} e' \overset{def}{\Leftrightarrow} \exists\, E, a\,.\, \left(\begin{array}{l} e = E[\delta_{\mathrm{r}}(a)] \\ e' = E[\delta_{\mathrm{v}}(\mathbf{T}_{2\to1}(a))] \\ \mathbf{T}_{2\to1}(a) \in \mathbf{V} \end{array} \right).$$

Remarquons que cette relation est déterministe : en conséquence, à chaque programme e à deux niveaux, on peut associer l'unique trace suivant la relation $\overset{rv}{\to}$ commençant en e. Pour montrer que le système d'équations est équivalent à un système gardé, il suffit de montrer que la relation $\overset{rv}{\to}$ termine.
On montre par récurrence sur le terme e que si e est un programme, alors la trace suivant $\overset{rv}{\to}$ associée à e est finie.
Soit e un programme.
Si e est une valeur, aucune réduction n'est possible.
Supposons $e = E[r]$, ou E est un contexte de réduction et r un radical, et examinons les différents cas pour E.
• $E = -$
S'il existe un programme a tel que r vaut $\delta_{\mathrm{r}}(a)$ et $\mathbf{T}_{2\to1}(a) \in \mathbf{V}$, alors e se réduit par $\overset{rv}{\to}$ en $\delta_{\mathrm{v}}(\mathbf{T}_{2\to1}(a))$, qui ne se réduit pas.
Sinon, e ne se réduit pas.
• $E = v_1\, E_2$
Par l'hypothèse de récurrence, la trace associée à $E_2[r]$ suivant $\overset{rv}{\to}$ se termine par un programme e_2. On constate que la trace associée à e se termine alors par $v_1\, e_2$.
• $E = E_1\, e_2$
Par l'hypothèse de récurrence, la trace associée à $E_1[r]$ suivant $\overset{rv}{\to}$ se termine par un programme e_1. Il s'ensuit que e se réduit par $\overset{rv}{\to}$ en $e_1\, e_2$.
Si e_1 n'est pas une valeur, alors $e_1\, e_2$ ne peut pas se réduire, puisque sinon, e_1 se réduirait.
Si e_1 est une valeur, on raisonne comme précédemment. Par l'hypothèse de récurrence, la trace associée à e_2 suivant $\overset{rv}{\to}$ se termine par un programme e_2'. On constate que la trace associée à e se termine alors par $e_1\, e_2'$.
Le système d'équations est donc équivalent à un système gardé. Le système équivalent est quasi-uniforme et compatible avec le système d'inférence définissant les résultats d'exécution (cf. tab.2.11 (p. 136)) pour le λ-calcul simple : d'après la proposition 1.2.4 (p. 75)), pour tout couple (e, ρ) tel que $e \Downarrow_2 \rho$, la valeur de la solution en $X_{(e,\rho)}$ est une preuve de $\mathbf{T}_{2\to1}(e) \Downarrow \mathbf{T}_{2\to1}(\rho)$.
⌞

Montrons maintenant que la relation d'évaluation $\Downarrow_2$ est déterministe.

3.1.2 Proposition (Déterminisme de l'évaluation)
Soient e un programme à deux niveaux, ρ_1 et ρ_2 deux résultats d'évaluation.

Si e s'évalue en ρ_1 et en ρ_2, alors les résultats ρ_1 et ρ_2 sont égaux :

$$\frac{e \Downarrow_2 \rho_1 \quad e \Downarrow_2 \rho_2}{\rho_1 = \rho_2}.$$

Démonstration
Supposons $e \Downarrow_2 \rho_1$ et $e \Downarrow_2 \rho_2$.
Montrons tout d'abord que si ρ_1 est différent de $\bot$, alors $\rho_1 = \rho_2$.
Supposons $\rho_1 \neq \bot$. On remarque tout d'abord que le jugement $e \Downarrow_2 \rho_1$ est validé par une preuve bien fondée, utilisant les règles [VAL], [β+], [PRIV/β+], [PRIV/CONV+] et [PRIV/DIV]. Pour montrer l'égalité, il suffit alors de raisonner par récurrence structurelle sur la preuve bien fondée de $e \Downarrow_2 \rho_1$: c'est immédiat.
Supposons $\rho_1 = \bot$. Raisonnons par l'absurde. Si $\rho_2 \neq \bot$, par le résultat précédent, $\rho_1 = \rho_2$, contradiction.
⌞

Enfin, montrons que la relation d'évaluation est totale.

3.1.3 Proposition (Totalité de l'évaluation)
Tout programme admet un résultat d'exécution :

$$\forall\, e \in \Lambda_2^0\,.\ \exists\, \rho\,.\ e \Downarrow_2 \rho\,.$$

Démonstration
Répartissons les programmes en deux ensembles, d'une part ceux qui admettent un résultat d'exécution différent de $\bot$, d'autre part ceux qui n'en admettent pas. Soit D ce dernier ensemble :

$$D \stackrel{def}{=} \{e \in \Lambda_2^0 \mid \neg(\exists\, \rho\,.\ \rho \neq \bot \wedge e \Downarrow_2 \rho)\}\,.$$

Pour tout programme e de D, on va construire une preuve de

$$e \Downarrow_2 \bot$$

en résolvant un système d'équations récursives, d'inconnues $(X_e)_{e \in D}$, ce qui permettra de conclure.
Soit e un programme de D. Nécessairement, e peut se décomposer en un radical dans un contexte de réduction. Le radical peut être d'une des trois formes suivantes :

$$(\lambda x\, a)\, v\,,$$
$$\delta_{\mathrm{v}}(\lambda x\, a)\, v\,,$$
$$\delta_{\mathrm{r}}(a)\,.$$

Dans le troisième cas, nécessairement $\mathbf{T}_{2\to 1}(a)$ est convergent, sinon e s'évaluerait en $\delta_{\mathrm{v}}(\bot)$.
Décrivons maintenant les équations du système, suivant les différentes décompositions possibles, correspondant respectivement aux règles d'inférence [$\beta-$], [PRIV/$\beta-$] et [PRIV/CONV–] :

$$\begin{aligned}
X_{E[(\lambda x\, a)\, v]} &= \frac{-\, X_{E[a[v/x]]}}{E[(\lambda x\, a)\, v] \;\Downarrow_2\; \bot}, \\
X_{E[\delta_{\mathrm{v}}(\lambda x\, a)\, v]} &= \frac{-\, X_{E[\delta_{\mathrm{r}}(a[v/x])]}}{E[\delta_{\mathrm{v}}(\lambda x\, a)\, v] \;\Downarrow_2\; \bot}, \\
X_{E[\delta_{\mathrm{r}}(a)]} &= \frac{-\, X_{E[\delta_{\mathrm{v}}(\lambda x\, b)]}}{E[\delta_{\mathrm{r}}(a)] \;\Downarrow_2\; \bot} \quad (\mathbf{T}_{2\to 1}(a) \;\Downarrow\; \lambda x\, b)\,.
\end{aligned}$$

Vérifions que les équations sont bien formées. Elles ont toutes la forme

$$X_e = \frac{-\, X_{e'}}{e \;\Downarrow_2\; \bot}.$$

Dans chaque cas, on peut constater que si e' s'évalue en un résultat différent de $\bot$, il en est de même de e, soit en contraposant, si e appartient à D, alors e' aussi.
Enfin, ce système d'équations est quasi-uniforme et compatible avec le système d'inférence définissant les résultats d'exécution (cf. tab.3.1 (p. 196)) : d'après la proposition 1.2.4 (p. 75)), pour tout programme e de D, la valeur de la solution en X_e est une preuve de $e \;\Downarrow_2\; \bot$.
⌞

Nous allons maintenant interpréter le niveau privé abstraitement. Précisément, nous allons interpréter tout terme $\delta_{\mathrm{r}}(e)$ par δ_{r}, et tout terme $\delta_{\mathrm{v}}(\lambda x\, e)$ par δ_{v}. La syntaxe du langage abstrait est donc définie par la grammaire suivante :

$$e \quad ::= \quad x \mid \lambda x\, e \mid e\, e \mid \delta_{\mathrm{r}} \mid \delta_{\mathrm{v}}\,.$$

La sémantique opérationnelle abstraite se définit comme précédemment en distinguant deux niveaux, par une relation de réduction pour le code public, par une relation d'évaluation maintenant triviale pour le code privé. Nous reprenons la description réalisée pour le langage concret à deux niveaux.
On ajoute à la définition habituelle d'une substitution l'invariance de δ_{r} et

δ_{v} ; si σ est une substitution, on a ainsi

$$\begin{aligned} \delta_{\mathrm{r}}[\sigma] &= \delta_{\mathrm{r}}\,, \\ \delta_{\mathrm{v}}[\sigma] &= \delta_{\mathrm{v}}\,. \end{aligned}$$

Une *valeur* est ou bien une abstraction close publique, ou bien la valeur abstraite δ_{v} :

$$v \quad ::= \quad \lambda x\, e \mid \delta_{\mathrm{v}}\,.$$

On note $\mathbf{V}_{\mathrm{A}}$ l'ensemble des valeurs.
Un *radical* est ou bien l'application d'une valeur à une autre valeur, ou bien le radical abstrait δ_{r} :

$$r \quad ::= \quad v\, v \mid \delta_{\mathrm{r}}\,.$$

Un *contexte de réduction* se construit autour de la place, en appliquant des valeurs, ou en passant des arguments :

$$E \quad ::= \quad - \mid v\, E \mid E\, e\,,$$

où e est un programme.
Comme d'habitude, tout programme peut se décomposer de manière unique en une valeur ou un radical placé dans un contexte de réduction : si e est un programme, ou bien c'est une valeur, ou bien il existe un unique radical r et un unique contexte de réduction E tel que $e = E[r]$ ($E[r]$ abrégeant $E[r/-]$).
Enfin, un *résultat d'évaluation* est soit une valeur, soit la divergence ; on distingue deux valeurs de divergence, une par niveau, notées $\bot$ pour le code public, $\delta_{\mathrm{v}}(\bot)$ pour le code privé, entendue comme la conversion publique de la divergence du code privé :

$$\rho \quad ::= \quad v \mid \bot \mid \delta_{\mathrm{v}}(\bot)\,.$$

La relation d'évaluation, notée $\Downarrow_{\mathrm{A}}$, qui associe à un programme un résultat d'évaluation, est définie par le système d'inférence de la table 3.3 (p. 202), pour lequel une interprétation mixte est utilisée ; il s'agit de l'interprétation abstraite des règles du système d'inférence de la table 3.1 (p. 196), définissant la sémantique opérationnelle concrète. L'interprétation abstraite rend la relation d'évaluation non déterministe. Ainsi, le radical abstrait δ_{r} s'évalue par la règle [PRIV/CONV+] (et la règle [VAL]) en δ_{v}, et par la règle [PRIV/DIV] en $\delta_{\mathrm{v}}(\bot)$; il en est de même de l'application $\delta_{\mathrm{v}}\, v$, qui s'évalue comme δ_{r} par la règle [PRIV/β+]. Les deux propositions suivantes permettent de préciser la nature du non-déterminisme. Si l'on s'intéresse aux seules valeurs, la relation d'évaluation est déterministe.

$$\frac{\emptyset}{v \Downarrow_{\mathrm{A}} v}\,[\mathrm{VAL}] \quad (v \in \mathbf{V}_{\mathrm{A}})$$

$$\frac{+\,E[a[v/x]] \Downarrow_{\mathrm{A}} \rho}{E[(\lambda x\, a)\, v] \Downarrow_{\mathrm{A}} \rho}\,[\beta+] \quad \left(\begin{array}{l} \rho \neq \bot \\ v \in \mathbf{V}_{\mathrm{A}} \end{array}\right)$$

$$\frac{-\,E[a[v/x]] \Downarrow_{\mathrm{A}} \bot}{E[(\lambda x\, a)\, v] \Downarrow_{\mathrm{A}} \bot}\,[\beta-] \quad (v \in \mathbf{V}_{\mathrm{A}})$$

$$\frac{+\,E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} \rho}{E[\delta_{\mathrm{v}}\, v] \Downarrow_{\mathrm{A}} \rho}\,[\mathrm{PRIV}/\beta+] \quad \left(\begin{array}{l} \rho \neq \bot \\ v \in \mathbf{V}_{\mathrm{A}} \end{array}\right)$$

$$\frac{-\,E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} \bot}{E[\delta_{\mathrm{v}}\, v] \Downarrow_{\mathrm{A}} \bot}\,[\mathrm{PRIV}/\beta-] \quad (v \in \mathbf{V}_{\mathrm{A}})$$

$$\frac{+\,E[\delta_{\mathrm{v}}] \Downarrow_{\mathrm{A}} \rho}{E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} \rho}\,[\mathrm{PRIV/CONV}+] \quad (\rho \neq \bot)$$

$$\frac{-\,E[\delta_{\mathrm{v}}] \Downarrow_{\mathrm{A}} \bot}{E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} \bot}\,[\mathrm{PRIV/CONV}-]$$

$$\frac{\emptyset}{E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)}\,[\mathrm{PRIV/DIV}]$$

Tab. 3.3 – Sémantique opérationnelle abstraite du langage à deux niveaux

3.1.4 Proposition (Évaluation abstraite : Déterminisme pour les valeurs)

Soient e un programme abstrait à deux niveaux, v_1 et v_2 deux valeurs abstraites. Si e s'évalue en v_1 et en v_2, alors les valeurs v_1 et v_2 sont égales :

$$\frac{e \Downarrow_{\mathrm{A}} v_1 \quad e \Downarrow_{\mathrm{A}} v_2}{v_1 = v_2} \quad (v_1, v_2 \in \boldsymbol{V}_{\mathrm{A}})\,.$$

Démonstration
On remarque tout d'abord que les jugements $e \Downarrow_{\mathrm{A}} v_1$ et $e \Downarrow_{\mathrm{A}} v_2$ sont validés par des preuves bien fondées, utilisant les règles [VAL], [β+], [PRIV/β+] et [PRIV/CONV+]. Pour montrer l'égalité, il suffit de raisonner par récurrence structurelle sur la preuve bien fondée de $e \Downarrow_{\mathrm{A}} v_1$: c'est immédiat.
⌞

Un programme abstrait ou bien converge (vers une valeur), ou bien diverge publiquement, ces deux possibilités s'excluant.

3.1.5 Proposition (Évaluation abstraite : Déterminisme et totalité hors divergence en privé)

Soit e un programme abstrait à deux niveaux. Alors e converge vers une valeur si et seulement si e ne diverge pas publiquement :

$$\exists\, v \in \boldsymbol{V}_{\mathrm{A}}\,.\; e \Downarrow_{\mathrm{A}} v \Leftrightarrow \neg(e \Downarrow_{\mathrm{A}} \bot)\,.$$

Démonstration
• $\exists\, v \in \mathbf{V}_{\mathrm{A}}\,.\; e \Downarrow_{\mathrm{A}} v \Rightarrow \neg(e \Downarrow_{\mathrm{A}} \bot)$
Soit v une valeur telle que $e \Downarrow_{\mathrm{A}} v$. Il est facile de montrer par récurrence structurelle sur la preuve bien fondée de $e \Downarrow_{\mathrm{A}} v$ que $\neg(e \Downarrow_{\mathrm{A}} \bot)$.
• $\neg(e \Downarrow_{\mathrm{A}} \bot) \Rightarrow \exists\, v \in \mathbf{V}_{\mathrm{A}}\,.\; e \Downarrow_{\mathrm{A}} v$
On contrapose. Soit D l'ensemble des programmes e tels que pour toute valeur v, on ait $\neg(e \Downarrow_{\mathrm{A}} v)$. Pour tout programme e de D, on construit une preuve de

$$e \Downarrow_{\mathrm{A}} \bot$$

en résolvant un système d'équations récursives, d'inconnues $(X_e)_{e \in D}$.
Comme par définition, tout programme de D se décompose en un radical dans un contexte de réduction, nous pouvons décrire les équations de ce système suivant les différentes décompositions possibles, correspondant aux

règles d'inférence [$\beta-$], [PRIV/$\beta-$] et [PRIV/CONV–] :

$$X_{E[(\lambda x\, a)\, v]} = \frac{-\, X_{E[a[v/x]]}}{E[(\lambda x\, a)\, v] \Downarrow_{\mathrm{A}} \bot},$$

$$X_{E[\delta_{\mathrm{v}}\, v]} = \frac{-\, X_{E[\delta_{\mathrm{r}}]}}{E[\delta_{\mathrm{v}}\, v] \Downarrow_{\mathrm{A}} \bot},$$

$$X_{E[\delta_{\mathrm{r}}]} = \frac{-\, X_{E[\delta_{\mathrm{v}}]}}{E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} \bot}.$$

Vérifions que les équations sont bien formées. Elles ont toutes la forme

$$X_e = \frac{-\, X_{e'}}{e \Downarrow_{\mathrm{A}} \bot}.$$

Dans chaque cas, on peut constater que si e' s'évalue en une valeur, il en est de même de e, soit en contraposant, si e appartient à D,alors e' aussi.
Enfin, ce système d'équations est quasi-uniforme et compatible avec le système d'inférence définissant les résultats d'exécution abstraits (cf. tab.3.3 (p. 202)) : d'après la proposition 1.2.4 (p. 75)), pour tout programme e de D, la valeur de la solution en X_e est une preuve de $e \Downarrow_{\mathrm{A}} \bot$.
⌞

Ces deux dernières propositions permettent de distinguer différents cas pour l'observation des résultats d'un programme abstrait à deux niveaux. Avant de les énoncer, introduisons une notation utile pour la relation d'évaluation, permettant de décrire l'ensemble des résultats d'évaluation de tout programme abstrait e :

$$e \Downarrow_{\mathrm{A}} \{\rho_0, \ldots, \rho_{n-1}\} \stackrel{def}{\Leftrightarrow} \forall\, \rho\,.\ e \Downarrow_{\mathrm{A}} \rho \Leftrightarrow (\exists\, i < n\,.\ \rho = \rho_i)\,.$$

Étant donné un programme abstrait e, il est facile de voir que les résultats de son exécution vérifient une et une seule des trois propriétés suivantes :

– [$\top$] : le programme *converge uniformément* (vers une unique valeur) :

$$\exists\, !\, v \in \mathbf{V}_{\mathrm{A}}\,.\ e \Downarrow_{\mathrm{A}} \{v\}\,,$$

– [$\bot$] : le programme *diverge uniformément* :

$$(e \Downarrow_{\mathrm{A}} \{\bot\}) \vee (e \Downarrow_{\mathrm{A}} \{\bot, \delta_{\mathrm{v}}(\bot)\})\,,$$

– [Δ] : le programme *dépend* de δ_{r} (ou converge et diverge en privé) :

$$\exists\, v \in \mathbf{V}_{\mathrm{A}}\,.\ e \Downarrow_{\mathrm{A}} \{v, \delta_{\mathrm{v}}(\bot)\}\,.$$

Donnons quelques exemples. N'importe quel programme convergent du λ-calcul simple, c'est-à-dire formé de code public uniquement, vérifie la propriété $[\top]$; de même, n'importe quel programme divergent du λ-calcul simple, vérifie la propriété $[\bot]$; si Ω est un tel programme, alors $(\lambda x\,\Omega)\,\delta_{\mathrm{r}}$ s'évalue en $\bot$ et $\delta_{\mathrm{v}}(\bot)$, et vérifie donc la propriété $[\bot]$; enfin, δ_{r}, qui s'évalue en δ_{v} et $\delta_{\mathrm{v}}(\bot)$ vérifie la propriété $[\Delta]$.
Par la suite, nous notons ainsi les trois propriétés précédentes :

- $e \Downarrow_{\mathrm{A}} [\top]$: le programme e vérifie la propriété $[\top]$,
- $e \Downarrow_{\mathrm{A}} [\bot]$: le programme e vérifie la propriété $[\bot]$,
- $e \Downarrow_{\mathrm{A}} [\Delta]$: le programme e vérifie la propriété $[\Delta]$.

Il nous reste à établir la correspondance entre le langage à deux niveaux et son *interprétation abstraite*.
Définissons tout d'abord par récurrence structurelle la fonction d'interprétation, notée $\mathbf{T}_{2\to\mathrm{A}}$, transformant tout programme à deux niveaux en un programme abstrait, où le code privé a été effacé :

$$\begin{aligned}
\mathbf{T}_{2\to\mathrm{A}}(x) &= x\,,\\
\mathbf{T}_{2\to\mathrm{A}}(\lambda x\, e) &= \lambda x\, \mathbf{T}_{2\to\mathrm{A}}(e)\,,\\
\mathbf{T}_{2\to\mathrm{A}}(f\, e) &= \mathbf{T}_{2\to\mathrm{A}}(f)\, \mathbf{T}_{2\to\mathrm{A}}(e)\,,\\
\mathbf{T}_{2\to\mathrm{A}}(\delta_{\mathrm{r}}(e)) &= \delta_{\mathrm{r}}\,,\\
\mathbf{T}_{2\to\mathrm{A}}(\delta_{\mathrm{v}}(\lambda x\, e)) &= \delta_{\mathrm{v}}\,.
\end{aligned}$$

On étend aussi cette fonction aux résultats d'exécution de la manière suivante :

$$\begin{aligned}
\mathbf{T}_{2\to\mathrm{A}}(\bot) &= \bot\,,\\
\mathbf{T}_{2\to\mathrm{A}}(\delta_{\mathrm{v}}(\bot)) &= \delta_{\mathrm{v}}(\bot)\,.
\end{aligned}$$

On remarque que la fonction d'interprétation commute avec les substitutions, puisque pour tous termes e_1 et e_2 et toute variable x, on a :

$$\mathbf{T}_{2\to\mathrm{A}}(e_1[e_2/x]) = \mathbf{T}_{2\to\mathrm{A}}(e_1)[\mathbf{T}_{2\to\mathrm{A}}(e_2)/x]\,.$$

Elle interprète aussi une valeur par une valeur, un contexte de réduction par un contexte de réduction et un radical par un radical. Par la suite, on utilisera ces propriétés sans mention.
La relation d'évaluation est préservée par l'interprétation abstraite du langage à deux niveaux, au sens donné par la proposition suivante.

3.1.6 Proposition (Interprétation abstraite : Préservation de la relation d'évaluation)

Soit e un programme à deux niveaux admettant ρ pour résultat d'exécution. Alors l'interprétation abstraite de e admet l'interprétation abstraite de ρ pour résultat d'exécution :

$$\frac{e \Downarrow_2 \rho}{\mathbf{T}_{2\to\mathrm{A}}(e) \Downarrow_\mathrm{A} \mathbf{T}_{2\to\mathrm{A}}(\rho)}.$$

Démonstration
Pour tout couple (e, ρ) tel que $e \Downarrow_2 \rho$, on construit une preuve de

$$\mathbf{T}_{2\to\mathrm{A}}(e) \Downarrow_\mathrm{A} \mathbf{T}_{2\to\mathrm{A}}(\rho)$$

en résolvant un système d'équations récursives, d'inconnues $(X_{(e,\rho)})_{(e,\rho)|e\Downarrow_2\rho}$. Pour faciliter une description générique des équations, on note ρ_+ un résultat d'exécution égal à une valeur ou à $\delta_\mathrm{v}(\bot)$, et ρ_- la valeur de divergence $\bot$. L'indice « + » ou « − » se réfère à l'interprétation mixte du système et indique la contrainte portant sur le profil de la preuve de la prémisse, « + » pour une preuve bien fondée, « − » pour une preuve infinie en profondeur. On décrit dans la table 3.4 (p. 207) l'équation associée à $X_{(e,\rho_*)}$ en fonction de la décomposition de e soit en une valeur, soit en un radical dans un contexte de réduction ; on a noté dans cette table $*$ un élément de la paire $\{+,-\}$. On vérifie aisément que ce système d'équations gardé et quasi-uniforme est compatible avec le système d'inférence définissant les résultats d'exécution (cf. tab. 3.3 (p. 202)) pour le langage abstrait : d'après la proposition 1.2.4 (p. 75)), pour tout couple (e, ρ) tel que $e \Downarrow_2 \rho$, la valeur de la solution en $X_{(e,\rho)}$ est une preuve de $\mathbf{T}_{2\to\mathrm{A}}(e) \Downarrow_\mathrm{A} \mathbf{T}_{2\to\mathrm{A}}(\rho)$.
⌞

Étudions une forme de réciproque, qui montre que toute évaluation abstraite est l'interprétation d'au moins une évaluation concrète : autrement dit, la relation d'évaluation est dense pour l'interprétation abstraite. Examinons trois cas, suivant le résultat de l'évaluation abstraite.
Commençons par les valeurs.

3.1.7 Proposition (Interprétation abstraite : Densité de la relation d'évaluation - Cas des valeurs)

Soit a un programme abstrait à deux niveaux convergent vers une valeur abstraite u :

$$a \Downarrow_\mathrm{A} u \quad (u \in \boldsymbol{V}_\mathrm{A}).$$

$$
\begin{aligned}
X_{(v,v)} &= \frac{\emptyset}{\mathbf{T}_{2\to\mathrm{A}}(v)\ \Downarrow_{\mathrm{A}}\ \mathbf{T}_{2\to\mathrm{A}}(v)} \quad (v \in \mathbf{V}_2) \\
X_{(E[(\lambda x\, a)\, v],\rho_*)} &= \frac{* X_{(E[a[v/x]],\rho_*)}}{\mathbf{T}_{2\to\mathrm{A}}(E[(\lambda x\, a)\, v])\ \Downarrow_{\mathrm{A}}\ \mathbf{T}_{2\to\mathrm{A}}(\rho_*)} \\
X_{(E[\delta_{\mathrm{v}}(\lambda x\, a)\, v],\rho_*)} &= \frac{* X_{(E[\delta_{\mathrm{r}}(a[v/x])],\rho_*)}}{\mathbf{T}_{2\to\mathrm{A}}(E[\delta_{\mathrm{v}}(\lambda x\, a)\, v])\ \Downarrow_{\mathrm{A}}\ \mathbf{T}_{2\to\mathrm{A}}(\rho_*)} \\
X_{(E[\delta_{\mathrm{r}}(a)],\rho_*)} &= \begin{cases} \dfrac{* X_{(E[\delta_{\mathrm{v}}(\lambda x\, b)],\rho_*)}}{\mathbf{T}_{2\to\mathrm{A}}(E[\delta_{\mathrm{r}}(a)])\ \Downarrow_{\mathrm{A}}\ \mathbf{T}_{2\to\mathrm{A}}(\rho_*)} & \text{si } \mathbf{T}_{2\to 1}(a)\ \Downarrow\ \lambda x\, b \\ \dfrac{\emptyset}{\mathbf{T}_{2\to\mathrm{A}}(E[\delta_{\mathrm{r}}(a)])\ \Downarrow_{\mathrm{A}}\ \delta_{\mathrm{v}}(\bot)} & \text{si } \mathbf{T}_{2\to 1}(a)\ \Downarrow\ \bot \end{cases}
\end{aligned}
$$

TAB. 3.4 – Dém. prop. 3.1.6

Alors il existe un programme concret à deux niveaux c ayant pour interprétation abstraite a et convergent vers une valeur concrète v :

$$c \Downarrow_2 v \quad (\mathbf{T}_{2\to\mathrm{A}}(c) = a, v \in \boldsymbol{V}_2)\,.$$

Démonstration
Soit $\mathrm{p}_{\max}$ le programme défini dans la démonstration de la proposition 2.2.5 (p. 144), et vérifiant la propriété suivante :

$$\forall\, v \in \mathbf{V}\,.\ \mathrm{p}_{\max}\, v \Downarrow \mathrm{p}_{\max}\,.$$

Ce programme est aussi une abstraction, qu'on note $\lambda z\, p$.
Associons à tout terme abstrait à deux niveaux a un terme concret, noté $T(a)$ et défini par récurrence sur a ainsi :

$$
\begin{aligned}
T(x) &= x\,, \\
T(\lambda x\, b) &= \lambda x\, T(b)\,, \\
T(a_1\, a_2) &= T(a_1)\, T(a_2)\,, \\
T(\delta_{\mathrm{r}}) &= \delta_{\mathrm{r}}(\mathrm{p}_{\max})\,, \\
T(\delta_{\mathrm{v}}) &= \delta_{\mathrm{v}}(\mathrm{p}_{\max})\,.
\end{aligned}
$$

Nous utiliserons par la suite quelques propriétés concernant T sans les mentionner. Tout d'abord, T commute avec les substitutions : pour tous termes

abstraits a_1 et a_2, et toute variable x, on a $T(a_1[a_2/x]) = T(a_1)[T(a_2)/x]$. Ensuite, l'image par T d'une valeur abstraite est une valeur concrète, et l'image d'un contexte de réduction abstrait est un contexte de réduction concret.
Nous allons montrer par récurrence sur la preuve de $a \Downarrow_{\mathrm{A}} u$ que

$$T(a) \Downarrow_2 T(u)\,.$$

Examinons les différents cas pour la règle concluant la preuve de

$$a \Downarrow_{\mathrm{A}} u\,.$$

- [VAL]
La conclusion est l'axiome $u \Downarrow_{\mathrm{A}} u$. Évidemment, $T(u)$, qui est une valeur, s'évalue en $T(u)$.
- $[\beta+]$
La preuve se termine par l'inférence suivante :

$$\frac{+\ E[b[v/x]] \Downarrow_{\mathrm{A}} u}{E[(\lambda x\, b)\, v] \Downarrow_{\mathrm{A}} u}\,.$$

On déduit de l'hypothèse de récurrence que $T(E[b[v/x]]) \Downarrow_2 T(u)$. Par la règle $[\beta+]$, on déduit que $T(E[(\lambda x\, b)\, v]) \Downarrow_2 T(u)$.
- [PRIV/β+]
La preuve se termine par l'inférence suivante :

$$\frac{+\ E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} u}{E[\delta_{\mathrm{v}}\, v] \Downarrow_{\mathrm{A}} u}\,.$$

On déduit de l'hypothèse de récurrence

$$T(E[\delta_{\mathrm{r}}]) \Downarrow_2 T(u)\,,$$

soit

$$T(E)[\delta_{\mathrm{r}}(\mathrm{p}_{\max})] \Downarrow_2 T(u)\,.$$

Nécessairement, ce jugement est déduit par la règle [PRIV/CONV+] du jugement

$$T(E)[\delta_{\mathrm{v}}(\mathrm{p}_{\max})] \Downarrow_2 T(u)\,.$$

Posons $v' = \mathbf{T}_{2\to 1}(T(v))$.
Par la règle [PRIV/CONV+], on déduit du jugement précédent le jugement

$$T(E)[\delta_{\mathrm{r}}(p[v'/z])] \Downarrow_2 T(u)\,,$$

puisque $\mathrm{p}_{\max}\, v'$, qui se réduit en $p[v'/z]$, s'évalue en $\mathrm{p}_{\max}$. Finalement, par la règle [PRIV/β+], on obtient

$$T(E)[\delta_{\mathrm{v}}(\mathrm{p}_{\max})\, T(v)] \Downarrow_2 T(u)\,.$$

• [PRIV/CONV+]
La preuve se termine par l'inférence suivante :

$$\frac{+\ E[\delta_{\mathrm{v}}] \Downarrow_{\mathrm{A}} u}{E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} u}\,.$$

On déduit de l'hypothèse de récurrence que $T(E)[\delta_{\mathrm{v}}(\mathrm{p}_{\max})] \Downarrow T(u)$. Par la règle [PRIV/CONV+], on obtient $T(E)[\delta_{\mathrm{r}}(\mathrm{p}_{\max})] \Downarrow T(u)$.
⌞

Venons-en à la divergence privée.

3.1.8 Proposition (Interprétation abstraite : Densité de la relation d'évaluation abstraite -) Cas de la divergence privée

Soit a un programme abstrait à deux niveaux divergent en privé :

$$a \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)\,.$$

Alors il existe un programme concret à deux niveaux c ayant pour interprétation abstraite a et divergent en privé :

$$c \Downarrow_2 \delta_{\mathrm{v}}(\bot) \quad (\mathbf{T}_{2\to\mathrm{A}}(c) = a)\,.$$

Démonstration
Soit $\mathrm{p}_{\min}$ un programme divergent, par exemple le programme $(\lambda x\, x\, x)\,(\lambda x\, x\, x)$. Associons à tout terme abstrait a un terme concret, noté $T(a)$ et défini par récurrence sur a ainsi :

$$\begin{aligned} T(x) &= x\,, \\ T(\lambda x\, b) &= \lambda x\, T(b)\,, \\ T(f\, a) &= T(f)\, T(a)\,, \\ T(\delta_{\mathrm{r}}) &= \delta_{\mathrm{r}}(\mathrm{p}_{\min})\,, \\ T(\delta_{\mathrm{v}}) &= \delta_{\mathrm{v}}(\lambda x\, \mathrm{p}_{\min})\,. \end{aligned}$$

Nous utiliserons par la suite quelques propriétés concernant T, sans les mentionner. Tout d'abord, T commute avec les substitutions : pour tous termes

abstraits a_1 et a_2, et toute variable x, on a $T(a_1[a_2/x]) = T(a_1)[T(a_2)/x]$. Ensuite, l'image par T d'une valeur abstraite est une valeur concrète, et l'image d'un contexte de réduction abstrait est un contexte de réduction concret.
Nous allons montrer par récurrence sur la preuve de $a \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)$ que

$$T(a) \Downarrow_2 \delta_{\mathrm{v}}(\bot)\,.$$

Examinons les différents cas pour la règle concluant la preuve de

$$a \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)\,.$$

- $[\beta+]$

La preuve se termine par l'inférence suivante :

$$\frac{+\ E[a[v/x]] \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)}{E[(\lambda x\, a)\, v] \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)}\,.$$

On déduit de l'hypothèse de récurrence que $T(E[a[v/x]]) \Downarrow_2 \delta_{\mathrm{v}}(\bot)$. Par la règle $[\beta+]$, on obtient $T(E[(\lambda x\, a)\, v]) \Downarrow_2 \delta_{\mathrm{v}}(\bot)$.

- [PRIV/$\beta+$]

La preuve se termine par l'inférence suivante :

$$\frac{+\ E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)}{E[\delta_{\mathrm{v}}\, v] \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)}\,.$$

On déduit de l'hypothèse de récurrence

$$T(E[\delta_{\mathrm{r}}]) \Downarrow_2 \delta_{\mathrm{v}}(\bot)\,,$$

soit

$$T(E)[\delta_{\mathrm{r}}(\mathrm{p}_{\min})] \Downarrow_2 \delta_{\mathrm{v}}(\bot)\,.$$

Par la règle [PRIV/$\beta+$], on obtient

$$T(E)[\delta_{\mathrm{v}}(\lambda x\, \mathrm{p}_{\min})\, T(v)] \Downarrow_2 \delta_{\mathrm{v}}(\bot)\,,$$

soit

$$T(E[\delta_{\mathrm{v}}\, v]) \Downarrow_2 \delta_{\mathrm{v}}(\bot)\,.$$

- [PRIV/CONV+]

La preuve se termine par l'inférence suivante :

$$\frac{+\ E[\delta_{\mathrm{v}}] \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)}{E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)}\,.$$

Par la règle [PRIV/DIV], on a directement $T(E)[\delta_{\mathrm{r}}(\mathrm{p}_{\min})] \Downarrow_2 \delta_{\mathrm{v}}(\bot)$, puisque $\mathrm{p}_{\min}$ diverge.
• [PRIV/DIV]
La preuve se termine par l'inférence suivante :

$$\frac{\emptyset}{E[\delta_{\mathrm{r}}] \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)}$$

Par la règle [PRIV/DIV], on a directement $T(E)[\delta_{\mathrm{r}}(\mathrm{p}_{\min})] \Downarrow_2 \delta_{\mathrm{v}}(\bot)$, puisque $\mathrm{p}_{\min}$ diverge.
⌞

Terminons par la divergence publique.

3.1.9 Proposition (Interprétation abstraite : Densité de la relation d'évaluation - Cas de la divergence publique)

Soit a un programme abstrait à deux niveaux divergent publiquement :

$$a \Downarrow_{\mathrm{A}} \bot .$$

Alors il existe un programme concret à deux niveaux c ayant pour interprétation abstraite a et divergent publiquement :

$$c \Downarrow_2 \bot \quad (\mathbf{T}_{2\to\mathrm{A}}(c) = a) .$$

Démonstration
On réutilise la fonction T définie au début de la démonstration de la proposition 3.1.7 (p. 206), vérifiant

$$\begin{aligned} T(\delta_{\mathrm{r}}) &= \delta_{\mathrm{r}}(\mathrm{p}_{\max}) , \\ T(\delta_{\mathrm{v}}) &= \delta_{\mathrm{v}}(\mathrm{p}_{\max}) . \end{aligned}$$

On rappelle que $\mathrm{p}_{\max}$ est le programme défini dans la démonstration de la proposition 2.2.5 (p. 144), qu'il vérifie

$$\forall\, v \in \mathbf{V} .\ \mathrm{p}_{\max}\, v \Downarrow \mathrm{p}_{\max} ,$$

et qu'il est égal à une abstraction, qu'on note $\lambda z\, p$.
Soit D l'ensemble des programmes abstraits divergents publiquement :

$$D \stackrel{def}{=} \{a \in \Lambda_A^0 \mid a \Downarrow_{\mathrm{A}} \bot\} .$$

Pour tout programme a de D, on va construire une preuve de

$$T(a) \Downarrow_2 \bot$$

en résolvant un système d'équations récursives, d'inconnues $(X_a)_{a\in D}$, ce qui permettra de conclure.
Décrivons les équations du système, suivant les différentes décompositions possibles d'un programme de D, qui correspondent respectivement aux règles d'inférence [$\beta-$], [PRIV/$\beta-$] et [PRIV/CONV–] :

$$\begin{aligned}
X_{E[(\lambda x\, a)\, v]} &= \frac{-\ X_{E[a[v/x]]}}{T(E[(\lambda x\, a)\, v]) \Downarrow_2 \bot}, \\
X_{E[\delta_{\mathrm{v}}\, v]} &= \frac{\dfrac{-\ X_{E[\delta_{\mathrm{v}}]}}{T(E)[\delta_{\mathrm{r}}(p[T(v)/z])] \Downarrow_2 \bot}}{T(E)[\delta_{\mathrm{v}}(\mathrm{p}_{\max})\, T(v)] \Downarrow_2 \bot}, \\
X_{E[\delta_{\mathrm{r}}(a)]} &= \frac{-\ X_{E[\delta_{\mathrm{v}}(\lambda x\, b)]}}{E[\delta_{\mathrm{r}}(a)] \Downarrow_2 \bot} \quad (\mathbf{T}_{2\to 1}(a) \Downarrow \lambda x\, b)\,.
\end{aligned}$$

Vérifions que les équations sont bien formées. Elles ont toutes la forme

$$X_a = \begin{array}{c} -\ X_{a'} \\ \vdots \\ T(a) \Downarrow_2 \bot \end{array}.$$

Dans chaque cas, on peut constater que si a' s'évalue en une valeur, il en est de même de a, soit en contraposant et en utilisant la proposition 3.1.5 (p. 203), si a appartient à D, alors a' aussi.
Enfin, ce système d'équations est équivalent à un système quasi-uniforme et compatible avec le système d'inférence définissant l'évaluation concrète (cf. tab.3.1 (p. 196)) : d'après la proposition 1.2.4 (p. 75)), pour tout programme a de D, la valeur de la solution en X_a est une preuve de $T(a) \Downarrow_2 \bot$.
⌞

Ainsi s'achève la description de la correspondance entre le langage à deux niveaux et son interprétation abstraite. Comme l'interprétation conserve l'évaluation[4], il apparaît que cette correspondance est forte : comme le dévoilent les trois dernières démonstrations, elle repose sur la capacité de

[4]Au sens où la relation d'évaluation est préservée par l'interprétation et dense pour elle.

l'évaluation abstraite de simuler deux évaluations concrètes particulières, celle du programme donnant lieu à une observation maximale et celle d'un programme divergent. C'est sur cette correspondance que nous nous fondons pour l'étude de la confidentialité.

3.1.2 Uniformité, dépendance, confidentialité

Nous commençons par nous intéresser à la question de savoir si le résultat observable d'un programme dépend de certains de ses sous-termes. À cette fin, nous modélisons un programme sous la forme d'un contexte possédant des places, où se greffent les sous-termes en question : les places rendent compte de la paramétrisation du programme. Par la suite, on appellera un contexte servant à une telle modélisation un programme paramétré, et ses places, les paramètres. Pour déterminer si un programme paramétré dépend de ses paramètres, il suffit de faire varier les termes remplaçant les paramètres et d'enregistrer les résultats d'exécution : si l'observation de ces résultats révèle des variations, c'est que le programme paramétré dépend de ses paramètres. Nous menons cette étude de la dépendance en utilisant le langage à deux niveaux décrit précédemment, le programme paramétré correspondant au code public, les termes remplaçant les paramètres au code privé ; l'interprétation abstraite nous permet de définir abstraitement la propriété de dépendance. Finalement, nous appliquons les résultats obtenus à la question de la confidentialité lors de l'exécution de code mobile.

Considérons deux nouveaux ensembles de places, $(\underline{\ }^{\mathrm{r}}_i)_{i>0}$ et $(\underline{\ }^{\mathrm{v}}_j)_{j>0}$; ces places seront appelées des paramètres. Dans un terme, un paramètre $\underline{\ }^{\mathrm{r}}_i$ peut être remplacée par n'importe quel terme, alors qu'un paramètre $\underline{\ }^{\mathrm{v}}_j$ peut uniquement être remplacé par une abstraction. On associe à toute expression C le *terme paramétré* aux n paramètres $(\underline{\ }^{\mathrm{r}}_i)_{i\in\{1,\dots,n\}}$ et aux p paramètres $(\underline{\ }^{\mathrm{v}}_i)_{i\in\{1,\dots,p\}}$, noté $C_{[n,p]}$: c'est l'application qui à tout n-uplet de termes $(e_1,\dots,e_n)$ et tout p-uplet d'abstractions $(f_1,\dots,f_p)$ associe le terme obtenu par la greffe simultanée dans C de e_i en chaque occurrence de $\underline{\ }^{\mathrm{r}}_i$ pour tout i et de f_j en chaque occurrence de $\underline{\ }^{\mathrm{v}}_j$ pour tout j. Un terme paramétré est donc un contexte particulier, qui utilise comme places les paramètres et restreint les possibilités de remplacement des paramètres. Un *programme paramétré* $C_{[n,p]}$ est un terme paramétré clos (les paramètres ne comptant pas comme des variables libres) dont le domaine de définition est restreint aux familles de termes $((e_i)_i,(f_j)_j)$ telles que $C[(e_i)_i,(f_j)_j]$ forme un programme. Par la suite, on suppose toujours tacitement que les familles de termes utilisées avec un programme paramétré appartiennent à son domaine de définition.

Nous sommes maintenant en mesure de définir la propriété de dépendance, comme négation d'une propriété d'uniformité.

3.1.10 Définition (Programme paramétré : Uniformité et dépendance)
Soit $C_{[n,p]}$ un programme paramétré. On dit que le programme paramétré $C_{[n,p]}$ est

- uniformément convergent*, ce qu'on note $C_{[n,p]} \Downarrow [\top]$, s'il converge quels que soient les termes remplaçant ses paramètres :*

$$C_{[n,p]} \Downarrow [\top] \stackrel{def}{\Leftrightarrow} \forall\, (e_i)_i, (f_j)_j \,.\; \exists\, v \in \boldsymbol{V}.\; C[(e_i)_i, (f_j)_j] \Downarrow v\,,$$

- uniformément divergent*, ce qu'on note $C_{[n,p]} \Downarrow [\bot]$, s'il diverge quels que soient les termes remplaçant ses paramètres :*

$$C_{[n,p]} \Downarrow [\bot] \stackrel{def}{\Leftrightarrow} \forall\, (e_i)_i, (f_j)_j \,.\; C[(e_i)_i, (f_j)_j] \Downarrow \bot\,.$$

On dit que le programme paramétré $C_{[n,p]}$ dépend de ses paramètres*, ce qu'on note $C_{[n,p]} \Downarrow [\Delta]$, s'il n'est ni uniformément convergent, ni uniformément divergent :*

$$C_{[n,p]} \Downarrow [\Delta] \stackrel{def}{\Leftrightarrow} \left(\begin{array}{l} \exists\, (e_i)_i, (f_j)_j, (e'_i)_i, (f'_j)_j, v \in \boldsymbol{V}. \\ \quad C[(e_i)_i, (f_j)_j] \Downarrow \bot \wedge C[(e'_i)_i, (f'_j)_j] \Downarrow v \end{array} \right).$$

Bien noter que les trois propriétés s'excluent mutuellement et couvrent l'ensemble des possibilités.

Il est simple de traduire tout terme paramétré en un terme paramétré à deux niveaux, les paramètres formant le code privé, le reste le code public. Précisément, à tout terme paramétré $C_{[n,p]}$ associons l'expression à deux niveaux $\mathbf{T}_{1\to 2}(C)$, définie par :

$$\mathbf{T}_{1\to 2}(C) \stackrel{def}{=} C[(\delta_{\mathrm{r}}(\underline{\ }^{\mathrm{r}}_i))_i, (\delta_{\mathrm{v}}(\underline{\ }^{\mathrm{v}}_j))_j]\,,$$

puis le terme paramétré à deux niveaux $\mathbf{T}_{1\to 2}(C)_{[n,p]}$.
Le lemme de l'effacement montre que cette traduction est complètement adéquate pour la convergence et la divergence, d'où son utilité.

3.1.11 Lemme (Traduction à deux niveaux : Adéquation complète pour la convergence et la divergence)
Soient $C_{[n,p]}$ un programme paramétré, $(e_1, \ldots, e_n)$ un n-uplet de termes et $(f_1, \ldots, f_p)$ un p-uplet d'abstractions. Alors on a les équivalences suivantes :

– $C[(e_i)_i, (f_j)_j]$ *converge si et seulement si* $\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]$ *converge :*

$$(\exists\, v \in \boldsymbol{V}.\ C[(e_i)_i, (f_j)_j] \Downarrow v) \Leftrightarrow \left(\begin{array}{l} \exists\, w \in \boldsymbol{V}_2\,. \\ \mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 w \end{array} \right),$$

– $C[(e_i)_i, (f_j)_j]$ *diverge si et seulement si* $\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]$ *diverge :*

$$(C[(e_i)_i, (f_j)_j] \Downarrow \bot) \Leftrightarrow \left(\begin{array}{l} \mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 \bot \\ \vee \\ \mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 \delta_{\mathrm{v}}(\bot) \end{array} \right).$$

Démonstration
$C[(e_i)_i, (f_j)_j]$ s'évalue en un unique résultat ρ_1 ; $\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]$ s'évalue en un unique résultat ρ_2. Par la proposition 3.1.1 (p. 195), on a $\mathbf{T}_{2\to 1}(\rho_2) = \rho_1$. Ainsi, si ρ_2 est une valeur, alors ρ_1 aussi, et si ρ_2 est égal à $\bot$ ou $\delta_{\mathrm{v}}(\bot)$, alors ρ_1 vaut $\bot$. On peut donc conclure.
⌞

Adaptons les définitions de l'uniformité et de la dépendance au langage à deux niveaux. Soit $C_{[n,p]}$ un programme paramétré. On dit que sa traduction à deux niveaux $\mathbf{T}_{1\to 2}(C)_{[n,p]}$ est

– *uniformément convergente*, ce qu'on note $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\top]$, si elle converge quels que soient les termes remplaçant ses paramètres :

$$\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\top] \overset{def}{\Leftrightarrow} \left(\begin{array}{l} \forall\, (e_i)_i, (f_j)_j\,. \\ \exists\, v \in \mathbf{V}_2\,.\ \mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 v \end{array} \right),$$

– *uniformément divergente*, ce qu'on note $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\bot]$, si elle diverge quels que soient les termes remplaçant ses paramètres :

$$\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\bot] \overset{def}{\Leftrightarrow} \left(\begin{array}{l} \forall\, (e_i)_i, (f_j)_j\,. \\ \quad \mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 \bot \\ \vee\, \mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 \delta_{\mathrm{v}}(\bot) \end{array} \right).$$

On dit que le programme paramétré à deux niveaux $\mathbf{T}_{1\to 2}(C)_{[n,p]}$ *dépend de ses paramètres*, ce qu'on note $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\Delta]$, s'il n'est ni uniformément convergent, ni uniformément divergent :

$$\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\Delta] \overset{def}{\Leftrightarrow} \left(\begin{array}{l} \exists\, (e_i)_i, (f_j)_j, (e'_i)_i, (f'_j)_j\,.\ \exists\, v \in \mathbf{V}_2\,. \\ \quad \left(\begin{array}{l} \mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 \delta_{\mathrm{v}}(\bot) \\ \vee\, \mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 \bot \end{array} \right) \\ \wedge\, (\mathbf{T}_{1\to 2}(C)[(e'_i)_i, (f'_j)_j] \Downarrow_2 v) \end{array} \right),$$

ce qui est équivalent à

$$\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\Delta] \overset{def}{\Leftrightarrow} \left(\begin{array}{l} \exists\, (e_i)_i, (f_j)_j, (e'_i)_i, (f'_j)_j \,.\ \exists\, v \in \mathbf{V}_2 \,. \\ \quad (\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 \delta_{\mathrm{v}}(\bot)) \\ \wedge\, (\mathbf{T}_{1\to 2}(C)[(e'_i)_i, (f'_j)_j] \Downarrow_2 v) \end{array} \right) .$$

En effet, s'il existe des familles $(e_i)_i$, $(f_j)_j$, $(e'_i)_i$ et $(f'_j)_j$, et une valeur à deux niveaux v telles que

$$(\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j] \Downarrow_2 \bot) \wedge (\mathbf{T}_{1\to 2}(C)[(e'_i)_i, (f'_j)_j] \Downarrow_2 v) ,$$

alors de la préservation de l'évaluation par l'interprétation abstraite (cf. prop. 3.1.6 (p. 205)), on déduit $\mathbf{T}_{2\to \mathrm{A}}(\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]) \Downarrow_{\mathrm{A}} \mathbf{T}_{2\to \mathrm{A}}(v)$ et $\mathbf{T}_{2\to \mathrm{A}}(\mathbf{T}_{1\to 2}(C)[(e'_i)_i, (f'_j)_j]) \Downarrow_{\mathrm{A}} \bot$; comme $\mathbf{T}_{2\to \mathrm{A}}(\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]) = \mathbf{T}_{2\to \mathrm{A}}(\mathbf{T}_{1\to 2}(C)[(e'_i)_i, (f'_j)_j])$, on obtient une contradiction d'après la proposition 3.1.5 (p. 203).
Bien noter là encore que les trois propriétés précédentes s'excluent mutuellement et couvrent l'ensemble des possibilités.
La proposition suivante montre que la traduction à deux niveaux est complètement adéquate pour les propriétés d'uniformité et de dépendance.

3.1.12 Proposition (Traduction à deux niveaux : Adéquation complète pour l'uniformité et la dépendance)

Soit $C_{[n,p]}$ un programme paramétré. Alors :

- *le contexte $C_{[n,p]}$ converge uniformément si et seulement si sa traduction à deux niveaux $\mathbf{T}_{1\to 2}(C)_{[n,p]}$ converge uniformément :*

$$C_{[n,p]} \Downarrow [\top] \Leftrightarrow \mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\top] ,$$

- *le contexte $C_{[n,p]}$ diverge uniformément si et seulement si sa traduction à deux niveaux $\mathbf{T}_{1\to 2}(C)_{[n,p]}$ diverge uniformément :*

$$C_{[n,p]} \Downarrow [\bot] \Leftrightarrow \mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\bot] ,$$

- *le contexte $C_{[n,p]}$ dépend de ses sous-termes si et seulement si sa traduction à deux niveaux $\mathbf{T}_{1\to 2}(C)_{[n,p]}$ dépend de ses sous-termes :*

$$C_{[n,p]} \Downarrow [\Delta] \Leftrightarrow \mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\Delta] .$$

Démonstration
Cette proposition découle directement de l'adéquation complète pour la convergence et la divergence, démontrées dans le lemme 3.1.11 (p. 214).
⌞

Nous allons maintenant donner une définition équivalente des propriétés d'uniformité et de dépendance en utilisant l'interprétation abstraite du langage à deux niveaux. Elle permet d'éviter le remplacement des paramètres par tous les termes possibles, puisque l'interprétation abstraite oublie le code privé.
Commençons par traduire un terme paramétré en un terme abstrait à deux niveaux, comme précédemment nous l'avons traduit en un terme concret. Précisément, à tout terme paramétré $C_{[n,p]}$ associons le terme abstrait à deux niveaux $\mathbf{T}_{1\to\mathrm{A}}(C)$, défini par :

$$\mathbf{T}_{1\to\mathrm{A}}(C) \stackrel{def}{=} C[(\delta_{\mathrm{r}})_i, (\delta_{\mathrm{v}})_j]\,.$$

On remarque que pour tout programme paramétré $C_{[n,p]}$, on a :

$$\mathbf{T}_{1\to\mathrm{A}}(C) = \mathbf{T}_{2\to\mathrm{A}}(\mathbf{T}_{1\to 2}(C))\,,$$

et que pour tout n-uplet de termes $(e_1, \ldots, e_n)$ et tout p-uplet d'abstractions $(f_1, \ldots, f_p)$, on a :

$$\mathbf{T}_{1\to\mathrm{A}}(C) = \mathbf{T}_{2\to\mathrm{A}}(\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j])\,.$$

Grâce à l'étroite correspondance entre le langage à deux niveaux et son interprétation abstraite, il est possible de définir les propriétés d'uniformité et d'indépendance en utilisant la traduction abstraite.

3.1.13 Théorème (Traduction abstraite : Adéquation complète pour l'uniformité et la dépendance)

Soit $C_{[n,p]}$ un programme paramétré. Alors :

- *le programme paramétré $C_{[n,p]}$ est uniformément convergent si et seulement si sa traduction abstraite à deux niveaux $\mathbf{T}_{1\to\mathrm{A}}(C)$ est uniformément convergente :*

$$C_{[n,p]} \Downarrow [\top] \Leftrightarrow \mathbf{T}_{1\to\mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\top]\,,$$

- *le programme paramétré $C_{[n,p]}$ est uniformément divergent si et seulement si sa traduction abstraite à deux niveaux $\mathbf{T}_{1\to\mathrm{A}}(C)$ est uniformément divergente :*

$$C_{[n,p]} \Downarrow [\bot] \Leftrightarrow \mathbf{T}_{1\to\mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\bot]\,,$$

- *le programme paramétré $C_{[n,p]}$ dépend de ses paramètres si et seulement si sa traduction abstraite à deux niveaux $\mathbf{T}_{1\to\mathrm{A}}(C)$ dépend de δ_{r} :*

$$C_{[n,p]} \Downarrow [\Delta] \Leftrightarrow \mathbf{T}_{1\to\mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\Delta]\,.$$

Démonstration
Considérons un programme paramétré $C_{[n,p]}$. Du fait de l'équivalence établie par la proposition 3.1.12 (p. 216), il suffit de montrer que pour tout d appartenant à $\{\bot, \top, \Delta\}$, on a

$$\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [d] \Leftrightarrow \mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [d]\,.$$

• $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\top] \Rightarrow \mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\top]$
Supposons $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\top]$.
Si $\mathbf{T}_{1\to \mathrm{A}}(C)$ diverge, c'est-à-dire si $\mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} \bot$ ou $\mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} \delta_{\mathrm{v}}(\bot)$, alors par densité (cf. prop. 3.1.8 (p. 209) et 3.1.9 (p. 211)), il existe un n-uplet de termes $(e_i)_i$ et un p-uplet d'abstractions $(f_j)_j$ tels que le programme concret à deux niveaux $\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]$ diverge, contradiction.
Finalement, on a bien $\mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\top]$.
• $\mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\top] \Rightarrow \mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\top]$
Supposons $\mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\top]$.
Soient $(e_i)_i$ un n-uplet de termes et $(f_j)_j$ un p-uplet d'abstractions.
Si $\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]$ diverge, par préservation de l'évaluation (cf. prop. 3.1.6 (p. 205)), il en est de même de $\mathbf{T}_{1\to \mathrm{A}}(C)$, contradiction.
Finalement, $\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]$ converge.
• $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\bot] \Rightarrow \mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\bot]$
Supposons $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\bot]$.
Si $\mathbf{T}_{1\to \mathrm{A}}(C)$ converge vers une valeur, alors par densité (cf. prop. 3.1.7 (p. 206)), il existe un n-uplet de termes $(e_i)_i$ et un p-uplet d'abstractions $(f_j)_j$ tels que le programme concret à deux niveaux $\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]$ converge, contradiction.
Finalement, on a bien $\mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\bot]$.
• $\mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\bot] \Rightarrow \mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\bot]$
Supposons $\mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\bot]$.
Soient $(e_i)_i$ un n-uplet de termes et $(f_j)_j$ un p-uplet d'abstractions.
Si $\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]$ converge vers une valeur, par préservation de l'évaluation (cf. prop. 3.1.6 (p. 205)), il en est de même de $\mathbf{T}_{1\to \mathrm{A}}(C)$, contradiction. Finalement, $\mathbf{T}_{1\to 2}(C)[(e_i)_i, (f_j)_j]$ diverge.
• $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\Delta] \Leftrightarrow \mathbf{T}_{1\to \mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\Delta]$
Il suffit d'utiliser les deux équivalences précédentes et de se rappeler que $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\Delta]$ est la négation de la disjonction de $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\top]$ et de $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_2 [\bot]$, tout comme $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_{\mathrm{A}} [\Delta]$ est la négation de la disjonction de $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_{\mathrm{A}} [\top]$ et de $\mathbf{T}_{1\to 2}(C)_{[n,p]} \Downarrow_{\mathrm{A}} [\bot]$.
⌞

Nous allons maintenant appliquer cette définition abstraite de l'unifor-

mité et de la dépendance au problème de la confidentialité pour le code mobile.
Rappelons que nous modélisons le code mobile par un programme mobile, précisément une abstraction dont la variable représente l'environnement, et l'environnement local par un programme local. Pour rendre compte des flux d'informations, on suppose que le programme local est paramétré, le terme remplaçant ce paramètre représentant une ressource dont le contenu doit rester confidentiel. Introduisons les notations correspondantes :

- $\lambda x\, \mathsf{M}[x]$ est le programme mobile, M étant un programme paramétré possédant un unique paramètre, à remplacer par l'environnement,
- $\mathsf{L}_{[\ ]}$ est le programme local, programme paramétré possédant un unique paramètre, à remplacer par la ressource à protéger.

Étant donné un terme δ représentant la ressource à protéger, l'exécution du code mobile dans l'environnement local se modélise par l'exécution du programme

$$(\lambda x\, \mathsf{M}[x])\, \mathsf{L}[\delta]\,.$$

Comme nous l'avons vu en introduction (cf. p. 12), la propriété de confidentialité exprime que pour tout code mobile, le résultat observable du code mobile dans l'environnement local ne dépend pas du paramètre local. Rappelons qu'on observe du résultat sa divergence ou sa convergence.

3.1.14 Définition (Environnement : Confidentialité)
L'environnement $\mathsf{L}_{[\ ]}$ *garantit la* confidentialité *de la ressource si pour tout code mobile* M*, pour tout couple de termes* (δ_1, δ_2)*, on a l'équivalence suivante :*

> *le programme* $(\lambda x\, \mathsf{M}[x])\, \mathsf{L}[\delta_1]$ *converge si et seulement si le programme* $(\lambda x\, \mathsf{M}[x])\, \mathsf{L}[\delta_2]$ *converge.*

La proposition suivante donne une définition équivalente.

3.1.15 Proposition (Confidentialité et équivalence contextuelle)
L'environnement $\mathsf{L}_{[\ ]}$ *garantit la confidentialité de la ressource si et seulement si pour tout couple de termes* (δ_1, δ_2)*, le programme local* $\mathsf{L}[\delta_1]$ *est équivalent contextuellement au programme local* $\mathsf{L}[\delta_2]$*.*

Démonstration
La condition donnée est évidemment suffisante. Montrons qu'elle est nécessaire.
Supposons que l'environnement $\mathsf{L}_{[\ ]}$ garantisse la confidentialité de la ressource. Soit (δ_1, δ_2) un couple de termes. De l'hypothèse, on déduit facilement que $\mathsf{L}[\delta_1]$ diverge si et seulement si $\mathsf{L}[\delta_2]$ diverge.

Distinguons deux cas, suivant que $\mathsf{L}[\delta_1]$ diverge ou non.
• $\mathsf{L}[\delta_1]$ diverge
$\mathsf{L}[\delta_2]$ diverge également et on a $\mathsf{L}[\delta_1] =_{\mathrm{C}} \mathsf{L}[\delta_2]$.
• $\mathsf{L}[\delta_1]$ converge
$\mathsf{L}[\delta_2]$ converge également. Soit $C_{[\ ]}$ un contexte. Comme $(\lambda x\, C[x])\, \mathsf{L}[\delta_1]$ se réduit en $C[\mathsf{L}[\delta_1]]$, et $(\lambda x\, C[x])\, \mathsf{L}[\delta_2]$ en $C[\mathsf{L}[\delta_2]]$, de l'hypothèse, on obtient que $C[\mathsf{L}[\delta_1]]$ converge si et seulement si $C[\mathsf{L}[\delta_2]]$ converge, d'où l'équivalence contextuelle.
⌞

En utilisant la sémantique observationnelle, la propriété de confidentialité devient équivalente à :

> pour tout couple de termes (δ_1, δ_2), l'observation du programme local $\mathsf{L}[\delta_1]$ est égale à celle du programme local $\mathsf{L}[\delta_2]$.

C'est cette dernière formulation qui nous amène à définir l'observation d'un programme abstrait. Comme l'observation d'un programme concret, cette observation peut être représentée par un arbre, dit *observationnel abstrait*. Ces arbres observationnels sont les termes respectant la signature concrète suivante. Une seule sorte est utilisée, notée o ; l'ensemble des étiquettes est égal à la paire $\{\bot, \top, \Delta\}$, $\bot$ représentant la divergence uniforme, $\top$ la convergence uniforme, Δ la dépendance relativement à δ_{r} (soit la convergence et la divergence en privé) ; l'ensemble des positions est engendré par l'ensemble des valeurs $\mathbf{V}$. Les étiquettes ont le profil suivant :

$$\begin{array}{l} \bot : () \to o\,, \\ \top : o^{\mathbf{V}} \to o\,, \\ \Delta : () \to o\,. \end{array}$$

Voyons comment associer à un programme abstrait[5] un arbre observationnel abstrait.

3.1.16 Définition et proposition (Observation d'un programme abstrait) *Il existe une unique application associant à tout programme abstrait à deux niveaux a, un arbre observationnel abstrait, noté* $\mathrm{Obs}^0_{\mathrm{A}}(a)$ *et vérifiant :*

$$\begin{array}{ll} (i) & \text{si } a \Downarrow_{\mathrm{A}} [\bot]\text{ , alors } \mathrm{Obs}^0_{\mathrm{A}}(a) = \bot\,, \\ (ii) & \text{si } a \Downarrow_{\mathrm{A}} [\top]\text{ , alors } \mathrm{Obs}^0_{\mathrm{A}}(a) = \top(\mathrm{Obs}^0_{\mathrm{A}}(a\, v))_{v \in \boldsymbol{V}}\,, \\ (iii) & \text{si } a \Downarrow_{\mathrm{A}} [\Delta]\text{ , alors } \mathrm{Obs}^0_{\mathrm{A}}(a) = \Delta\,. \end{array}$$

[5] Comme dans le cas concret, on aurait pu évidemment considérer plus généralement les termes abstraits : c'est cependant inutile ici compte tenu de l'application que nous avons en vue.

L'arbre observationnel abstrait $\mathrm{Obs}^0_{\mathrm{A}}(a)$ *est appelé l'*observation abstraite *de* a*.*

Démonstration
Les équations précédentes permettent de définir un système gardé d'équations récursives, d'où l'existence et l'unicité de l'application définissant l'observation abstraite.
⌞

Il est possible d'étendre la notion d'uniformité aux observations : l'observation d'un programme paramétré est uniforme si elle ne dépend pas de ses paramètres. Le lemme suivant permet de définir l'uniformité observationnelle à partir de l'observation abstraite.

3.1.17 Lemme (Traduction abstraite : Adéquation complète pour l'uniformité observationnelle)
Soient $C_{[n,p]}$ *un programme paramétré et* A *un arbre observationnel concret*[6]*. Alors l'observation abstraite de* $\mathbf{T}_{1\to\mathrm{A}}(C)$ *est égale à* A *si et seulement si pour tout* n*-uplet de termes* $(e_1,\ldots,e_n)$ *et tout* p*-uplet d'abstractions* $(f_1,\ldots,f_p)$*, l'observation de* $C[(e_i)_i,(f_j)_j]$ *est égale à* A *:*

$$(\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)) = A) \Leftrightarrow (\forall\,(e_i)_i,(f_j)_j\,.\ \mathrm{Obs}^0(C[(e_i)_i,(f_j)_j]) = A)\,.$$

Démonstration
• $\forall\, A, C_{[n,p]}\,.\ \left(\begin{array}{l}(\forall\,(e_i)_i,(f_j)_j\,.\ \mathrm{Obs}^0(C[(e_i)_i,(f_j)_j]) = A) \Rightarrow\\ (\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)) = A)\end{array}\right)$
Soit D l'ensemble des couples $(C_{[n,p]}, A)$, où $C_{[n,p]}$ est un programme paramétré et A un arbre observationnel concret, vérifiant :

$$\forall\,(e_i)_i,(f_j)_j\,.\ \mathrm{Obs}^0(C[(e_i)_i,(f_j)_j]) = A\,.$$

Pour tout couple $(C_{[n,p]}, A)$ de D, on construit une preuve de l'égalité entre $\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C))$ et A, en résolvant un système d'équations récursives, d'inconnues $(X_d)_{d\in D}$. Si $(C_{[n,p]}, A)$ appartient à D, l'équation associée est définie ainsi :

$$X_{(C_{[n,p]},A)} = \begin{cases} \dfrac{\emptyset}{(\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)), A)} & \text{si } A = \bot\,,\\[2ex] \dfrac{(X_{((C\,v)_{[n,p]},A_v)})_{v\in\mathbf{V}}}{(\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)), A)} & \text{si } A = \top(A_v)_{v\in\mathbf{V}}\,. \end{cases}$$

[6] Ses seules étiquettes sont donc $\bot$ et $\top$.

Il est facile de vérifier que dans le second cas, pour toute valeur v, le couple $((C\,v)_{[n,p]}, A_v)$ appartient à D.
Montrons que ce système quasi-uniforme est compatible avec le système d'inférence définissant l'égalité entre arbres observationnels.
Dans le premier cas, où $A = \bot$, le programme paramétré est uniformément divergent, d'où par le théorème 3.1.13 (p. 217), $\mathbf{T}_{1\to\mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\bot]$, puis $\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)) = \bot$. On obtient ainsi un axiome du système d'inférence.
Dans le second cas, où $A = \top(A_v)_{v\in\mathbf{V}}$, le programme paramétré est uniformément convergent, d'où par le théorème 3.1.13 (p. 217), $\mathbf{T}_{1\to\mathrm{A}}(C) \Downarrow_{\mathrm{A}} [\top]$, puis $\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)) = \top(\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)\,v))_{v\in\mathbf{V}}$. Comme $\mathbf{T}_{1\to\mathrm{A}}(C)\,v = \mathbf{T}_{1\to\mathrm{A}}(C\,v)$, la règle

$$\frac{\big((\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C\,v)), A_v)\big)_{v\in\mathbf{V}}}{(\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)), A)}$$

appartient bien au système d'inférence définissant l'égalité entre arbres observationnels.
Finalement, d'après la proposition 1.2.3 (p. 73), pour tout couple $(C_{[n,p]}, A)$ de D, la valeur de la solution en $X_{(C_{[n,p]},A)}$ est une preuve de l'égalité entre $\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C))$ et A.

- $\forall\, A, C_{[n,p]}\,.\ \left(\begin{array}{l}(\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)) = A) \Rightarrow \\ (\forall\,(e_i)_i, (f_j)_j\,.\ \mathrm{Obs}^0(C[(e_i)_i, (f_j)_j]) = A)\end{array}\right)$

Soit D l'ensemble des couples $(C_{[n,p]}, A)$, où $C_{[n,p]}$ est un programme paramétré et A un arbre observationnel concret, vérifiant :

$$\mathrm{Obs}^0_{\mathrm{A}}(\mathbf{T}_{1\to\mathrm{A}}(C)) = A\,.$$

Soient $(e_i)_i$ un n-uplet de termes et $(f_j)_j$ un p-uplet d'abstractions.
Pour tout couple $(C_{[n,p]}, A)$ de D, on construit une preuve de l'égalité entre $\mathrm{Obs}^0(C[(e_i)_i, (f_j)_j])$ et A, en résolvant un système d'équations récursives, d'inconnues $(X_d)_{d\in D}$. Si $(C_{[n,p]}, A)$ appartient à D, l'équation associée est définie ainsi :

$$X_{(C_{[n,p]},A)} = \begin{cases} \dfrac{\emptyset}{(\mathrm{Obs}^0(C[(e_i)_i, (f_j)_j]), A)} & \text{si } A = \bot\,, \\[2ex] \dfrac{(X_{((C\,v)_{[n,p]},A_v)})_{v\in\mathbf{V}}}{(\mathrm{Obs}^0(C[(e_i)_i, (f_j)_j]), A)} & \text{si } A = \top(A_v)_{v\in\mathbf{V}}\,. \end{cases}$$

Il est facile de vérifier que dans le second cas, pour toute valeur v, le couple $((C\,v)_{[n,p]}, A_v)$ appartient à D.

Montrons que ce système quasi-uniforme est compatible avec le système d'inférence définissant l'égalité entre arbres observationnels.
Dans le premier cas, où $A = \bot$, le programme abstrait à deux niveaux $\mathbf{T}_{1\to\mathrm{A}}(C)$ diverge uniformément, d'où par le théorème 3.1.13 (p. 217), le programme $C[(e_i)_i, (f_j)_j]$ diverge, puis $\mathrm{Obs}^0(C[(e_i)_i, (f_j)_j]) = \bot$. On obtient ainsi un axiome du système d'inférence.
Dans le second cas, où $A = \top(A_v)_{v\in\mathbf{V}}$, le programme abstrait à deux niveaux $\mathbf{T}_{1\to\mathrm{A}}(C)$ converge uniformément, d'où par le théorème 3.1.13 (p. 217), le programme $C[(e_i)_i, (f_j)_j]$ converge, ce qui donne pour son observation l'égalité $\mathrm{Obs}^0(C[(e_i)_i, (f_j)_j]) = \top(\mathrm{Obs}^0(C[(e_i)_i, (f_j)_j]\, v))_{v\in\mathbf{V}}$.
Comme $C[(e_i)_i, (f_j)_j]\, v$ est égal à $(C\, v)[(e_i)_i, (f_j)_j]$, la règle

$$\frac{\big((\mathrm{Obs}^0((C\, v)[(e_i)_i, (f_j)_j]), A_v)\big)_{v\in\mathbf{V}}}{(\mathrm{Obs}^0(C[(e_i)_i, (f_j)_j]), A)}$$

appartient bien au système d'inférence définissant l'égalité entre arbres observationnels.
Finalement, d'après la proposition 1.2.3 (p. 73), pour tout couple $(C_{[n,p]}, A)$ de D, la valeur de la solution en $X_{(C_{[n,p]},A)}$ est une preuve de l'égalité entre $\mathrm{Obs}^0(C[(e_i)_i, (f_j)_j])$ et A.
⌞

Nous sommes maintenant en mesure de caractériser exactement les environnements garantissant la confidentialité à l'aide de l'observation abstraite.

3.1.18 Théorème (Environnement : Critère abstrait de confidentialité)

L'environnement $\mathsf{L}_{[\ ]}$ *garantit la confidentialité de la ressource si et seulement si l'observation abstraite de* $\mathsf{L}[\delta_\mathrm{r}]$ *ne fait pas apparaître* Δ*, soit si et seulement si*

$$\forall V \in \boldsymbol{V}^*\,.\ \mathrm{Obs}^0_\mathrm{A}(\mathsf{L}[\delta_\mathrm{r}])(V) \neq \Delta\,.$$

Démonstration
Considérons la proposition exprimant que l'environnement $\mathsf{L}_{[\ ]}$ garantit la confidentialité de la ressource. Elle est successivement équivalente à :

$$\begin{array}{l}\forall\,(\delta_1, \delta_2)\,.\ \mathsf{L}[\delta_1] =_\mathrm{C} \mathsf{L}[\delta_2]\,,\\ \forall\,(\delta_1, \delta_2)\,.\ \mathrm{Obs}^0(\mathsf{L}[\delta_1]) = \mathrm{Obs}^0(\mathsf{L}[\delta_2])\,,\\ \forall V \in \mathbf{V}^*\,.\ \mathrm{Obs}^0_\mathrm{A}(\mathsf{L}[\delta_\mathrm{r}])(V) \neq \Delta \quad \text{(lem. 3.1.17)}\,.\end{array}$$

⌞

Ici s'achève notre analyse des flux d'informations, précisément de la confidentialité des ressources. Il a donc été montré qu'une analyse de l'environnement suffit pour déterminer s'il garantit la confidentialité des ressources qu'il protège, et de plus que cette analyse peut être menée abstraitement, sans aucune approximation. Le critère abstrait obtenu n'est évidemment pas décidable. S'il l'était, le problème de la la terminaison le serait pour le λ-calcul paresseux. En effet, considérons un programme e et appliquons la procédure de décision supposée à $(\lambda x\, \delta_{\mathrm{r}})\, e$. On constate que e termine si et seulement si l'observation abstraite de $\delta_{\mathrm{r}}\, e$ fait apparaître Δ.

3.2 Un critère de confinement

Nous nous intéressons maintenant au confinement. Nous allons enrichir notre langage par des références, permettant de manipuler des objets en mémoire. Nous définissons la syntaxe de ce langage, son système de types et sa sémantique opérationnelle, à partir d'une relation de réduction. Vient ensuite son annotation. La sémantique opérationnelle du langage annoté vise à préserver l'origine des opérateurs. C'est cette préservation qui permet de définir la frontière entre le code mobile et le code local. L'étude du confinement est menée après l'étude de la frontière. Enfin, nous proposons une solution pour contrôler les accès, fondée sur le contrôle de l'usage et le confinement des ressources dans le code local.

3.2.1 Manipuler des objets en mémoire

Comme annoncé, on ajoute au λ-calcul quelques primitives permettant de manipuler des objets en mémoire, et décrites ici à partir de leurs règles de typage :

- $\mathrm{unit} : \mathrm{Unit}$: unique valeur du type singleton Unit, appelée l'unité,
- $\dfrac{e : A}{\mathrm{ref}(e) : \mathrm{Ref}(A)}$: opérateur permettant la création d'une nouvelle référence, ayant pour contenu initial le résultat de l'évaluation de e,
- $l^A : \mathrm{Ref}(A)$: une référence l^A est identifiée par son identificateur l et a pour contenu une valeur de type A ;
- $\dfrac{e_1 : \mathrm{Ref}(A) \quad e_2 : A}{\mathrm{set}(e_1, e_2) : \mathrm{Unit}}$: opérateur permettant d'affecter e_2 au contenu de la référence e_1, et renvoyant la valeur unit,
- $\dfrac{e : \mathrm{Ref}(A)}{\mathrm{get}(e) : A}$: opérateur permettant de lire le contenu de la référence e.

$$
\begin{array}{rcll}
a & ::= & x & \text{(variable)} \\
 & | & \lambda x : A.\, a & \text{(abstraction)} \\
 & | & a\, a & \text{(application)} \\
 & | & \text{unit} & \text{(unité)} \\
 & | & l^A & \text{(référence)} \\
 & | & \text{ref}(a) & \text{(création de référence)} \\
 & | & \text{get}(a) & \text{(lecture)} \\
 & | & \text{set}(a, a) & \text{(écriture)} \\
 & & & \\
A & ::= & \text{Unit} & \text{(type singleton)} \\
 & | & A \to A & \text{(type fonctionnel)} \\
 & | & \text{Ref}(A) & \text{(type des références)}
\end{array}
$$

TAB. 3.5 – Syntaxe du langage avec références

Le langage est engendré par la grammaire décrite dans la table 3.5. Il utilise une notation à la Church[7]. On note $\mathbf{FV}(a)$ l'ensemble des variables libres de a et $\mathbf{Ref}(a)$ l'ensemble des références ayant une occurrence dans a. Un terme clos est un terme sans variable libre, un programme est un terme clos ne possédant pas de références :

$$
\begin{array}{rcl}
a \text{ terme clos} & \stackrel{def}{\Leftrightarrow} & \mathbf{FV}(a) = \emptyset\,, \\
a \text{ programme} & \stackrel{def}{\Leftrightarrow} & \mathbf{FV}(a) = \emptyset \wedge \mathbf{Ref}(a) = \emptyset\,.
\end{array}
$$

On impose une condition de bonne formation aux termes : pour tout terme a, il est supposé que la famille $(l)_{l^A \in \mathbf{Ref}(a)}$ est injective, autrement dit il est supposé que les références d'un terme peuvent être identifiées par leur identificateur. Par la suite, nous n'utilisons que des termes vérifiant cette condition et omettons de vérifier la préservation de cette condition, toujours évidente.
Le système de types de ce langage est décrit par la table 3.6. Il permet de

[7]Dans cette notation, la variable liée par une abstraction reçoit explicitement un type : cette notation s'oppose à celle de Curry, où le type est absent ; dans le cas de la notation de Church, chaque terme possède un type unique, alors que dans celle de Curry, il en possède un principal, les autres s'obtenant en remplaçant les variables de types par des types quelconques.

$$\frac{\emptyset}{\Gamma \vdash x \,:\, \Gamma(x)}\ [\text{intro-Var}] \quad (x \in \operatorname{dom}\Gamma)$$

$$\frac{\Gamma.(x \,:\, A) \vdash a \,:\, B}{\Gamma \vdash \lambda x : A.\, a \,:\, A \to B}\ [\text{intro-}\to]$$

$$\frac{\Gamma \vdash f \,:\, A \to B \quad \Gamma \vdash a \,:\, A}{\Gamma \vdash f\, a \,:\, B}\ [\text{élim-}\to]$$

$$\frac{\emptyset}{\Gamma \vdash \text{unit} \,:\, \text{Unit}}\ [\text{intro-Unit}]$$

$$\frac{\emptyset}{\Gamma \vdash l^A \,:\, \text{Ref}(A)}\ [\text{intro-Ref/loc}]$$

$$\frac{\Gamma \vdash a \,:\, A}{\Gamma \vdash \text{ref}(a) \,:\, \text{Ref}(A)}\ [\text{intro-Ref/new}]$$

$$\frac{\Gamma \vdash a \,:\, \text{Ref}(A)}{\Gamma \vdash \text{get}(a) \,:\, A}\ [\text{élim-Ref/ !}]$$

$$\frac{\Gamma \vdash a_1 \,:\, \text{Ref}(A) \quad \Gamma \vdash a_2 \,:\, A}{\Gamma \vdash \text{set}(a_1, a_2) \,:\, \text{Unit}}\ [\text{élim-Ref/ ?}]$$

TAB. 3.6 – Système de types du langage avec références

définir inductivement l'ensemble des jugements de typage valides, un jugement de typage étant de la forme $\Gamma \vdash a \,:\, A$, où Γ est un environnement de typage, a est un terme et A un type, ces derniers respectivement sujet et objet du jugement « a pour type » dans l'environnement Γ. Un environnement de typage sert au typage des variables libres ; c'est donc une fonction de l'ensemble des variables dans l'ensemble des types. Comme les références sont étiquetées par le type de leur contenu, il est inutile d'utiliser un environnement de typage pour les références. Donnons quelques propriétés classiques (parce qu'indispensables) du système de types.

3.2.1 Proposition (Propriétés du système de types)
Soit Γ un environnement de typage. Considérons un terme a de type A dans

l'environnement Γ *:*

$$\Gamma \vdash a : A\,.$$

Lemme d'affaiblissement
Considérons un environnement Γ' *prolongeant* Γ*. Alors le jugement de typage* $\Gamma' \vdash a : A$ *est valide.*

Lemme de la base
Considérons l'environnement $\Gamma_{|\mathbf{FV}(a)}$ *obtenu en restreignant* Γ *aux variables libres de* a*. Alors le jugement de typage* $\Gamma_{|\mathbf{FV}(a)} \vdash a : A$ *est valide.*

Unicité du type
Pour tout type B*, si* $\Gamma \vdash a : B$ *est valide, alors* $A = B$*.*

Démonstration
• Affaiblissement
Immédiat, par récurrence structurelle sur la preuve de $\Gamma \vdash a : A$.
• Base
On procède par récurrence structurelle sur la preuve de $\Gamma \vdash a : A$. Le seul cas intéressant concerne les abstractions.
Supposons $\Gamma \vdash \lambda x : A.\, b : A \to B$. Ce jugement se déduit de $\Gamma.(x : A) \vdash b : B$, et par hypothèse de récurrence, on a $(\Gamma.(x : A))_{|\mathbf{FV}(b)} \vdash b : B$.
Si x est une variable libre de b, on $(\Gamma.(x : A))_{|\mathbf{FV}(b)} = \Gamma_{|\mathbf{FV}(a)}.(x : A)$, sinon, $(\Gamma.(x : A))_{|\mathbf{FV}(b)} = \Gamma_{|\mathbf{FV}(a)}$. Dans tous les cas, en utilisant le lemme d'affaiblissement, on a $\Gamma_{|\mathbf{FV}(a)}.(x : A) \vdash b : B$, d'où on déduit $\Gamma_{|\mathbf{FV}(a)} \vdash \lambda x : A.\, b : A \to B$.
• Unicité
Immédiat, par récurrence structurelle sur la preuve de $\Gamma \vdash a : A$.
⌞

Le lemme de la base et l'unicité permettent de définir le type d'un terme clos typé sans ambiguïté. Par la suite, on utilisera de telles propriétés évidentes sans rappeler cette proposition.
La sémantique opérationnelle du langage avec références se définit à partir d'une relation de réduction. Donnons les définitions habituelles indispensables à cette définition.
Une substitution se définit comme d'habitude, en ajoutant cependant une contrainte de typage. Précisément, une substitution est un couple (Γ, σ) formé d'un environnement de typage Γ et d'une fonction σ de l'ensemble des variables dans l'ensemble des termes vérifiant :

– $\operatorname{dom}\Gamma = \operatorname{dom}\sigma$,
– pour toute variable x du domaine de σ, le terme $\sigma(x)$ a pour type $\Gamma(x)$ dans l'environnement Γ.

À tout terme e admettant un type dans l'environnement Γ, on associe un terme, noté $e[\sigma]$ ou $e[(\sigma(x)/x)_{x\in\operatorname{dom}\sigma}]$, défini comme d'habitude en remplaçant chaque variable libre de e appartenant au domaine de σ par son image par σ, et en évitant la capture des variables libres de cette dernière par les lieurs de e. Comme dans ces dernières notations, on omettra généralement de préciser l'environnement lors de l'utilisation d'une substitution : c'est qu'il pourra se déduire du contexte sans ambiguïté. Comme le montre la proposition suivante, une substitution préserve le type.

3.2.2 Proposition (Lemme des substitutions)
Soit (Γ, σ) une substitution. Considérons un terme a de type A dans l'environnement Γ et une variable x du domaine de Γ. Si le terme b a pour type $\Gamma(x)$ dans l'environnement Γ, alors le terme $a[b/x]$ a pour type A dans l'environnement Γ :

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : \Gamma(x)}{a[b/x] : A} \quad (x \in \operatorname{dom}\Gamma)\,.$$

Démonstration
On procède par récurrence structurelle sur la preuve de $\Gamma \vdash a : A$. C'est sans difficulté.
⌞

On doit pouvoir décomposer un terme clos en ou bien une valeur, ou bien un radical dans un contexte de réduction.
Une *valeur* est ou bien une abstraction close, ou bien la valeur unité, ou bien une référence, ce qu'on résume ainsi :

$$v \quad ::= \quad \lambda x : A.\, e \mid \mathrm{unit} \mid l^A\,,$$

où $\mathbf{FV}(\lambda x : A.\, e) = \emptyset$.
Un *radical* est ou bien l'application d'une abstraction close à une valeur, ou bien la création d'une référence, ou bien la lecture d'une référence, ou encore l'écriture d'une référence, ce qui donne :

$$r \quad ::= \quad (\lambda x : A.\, e)\, v \mid \mathrm{ref}(v) \mid \mathrm{get}(l^A) \mid \mathrm{set}(l^A, v)\,.$$

Un *contexte de réduction* se construit autour de la place[8] $-$ de la manière suivante :

$$E \quad ::= \quad - \mid E\, e \mid v\, E \mid \mathrm{ref}(E) \mid \mathrm{get}(E) \mid \mathrm{set}(E, e) \mid \mathrm{set}(v, E)\,,$$

[8] On rappelle que la place est une variable particulière, réservée à ce seul usage.

où $\mathbf{FV}(e) = \emptyset$. Si E est un contexte de réduction et e un terme, on note $E[e]$ la greffe[9] de e en la place $-$.
Un terme clos typé peut être décomposé d'une unique manière, comme l'indique la proposition suivante.

3.2.3 Proposition (Lemme de la décomposition)
Soit a un terme clos de type A.
Alors a n'est pas une valeur si et seulement s'il existe un contexte de réduction E et un radical r tels que a est égal à $E[r]$.
Dans ce cas, la décomposition en $E[r]$ est unique, le radical r est typé et le contexte de réduction E a pour type A, si on attribue à la place $-$ le type de r.

Démonstration
On vérifie facilement par simple examen que si $a = E[r]$, alors a n'est pas une valeur. On montre maintenant par récurrence sur a que si a est un terme clos typé qui n'est pas une valeur, alors il existe un contexte de réduction E et un radical r tels que $a = E[r]$ et tels que :
- si E' est un contexte de réduction et r' un radical tels que $a = E'[r']$, alors $E = E'$ et $r = r'$,
- r est typé,
- E a la même type que a si on attribue à la place $-$ le type de r.

• $a = x, a = \lambda x : A.\, b$
C'est trivialement vérifié.
• $a = a_1\, a_2$
a n'est pas une valeur. Supposons que a soit un terme clos typé, ce qui implique que a_1 et a_2 sont aussi clos et typés. Distinguons deux cas pour a_1, suivant que a_1 est une valeur ou non.
◦ a_1 est une valeur
Appliquons l'hypothèse de récurrence à a_2. Deux cas se présentent encore.
Si a_2 est une valeur, alors a est un radical, puisque par les règles de typage, a_1 ne peut être qu'une abstraction, et on vérifie facilement que $E = -$ et $r = a$ conviennent.
Sinon, il existe un contexte de réduction E_2 et un radical r_2 tels que $a_2 = E_2[r_2]$ et vérifiant les autres propriétés. On vérifie facilement que $E = a_1\, E_2$ et $r = r_2$ conviennent.
◦ a_1 n'est pas une valeur
Appliquons l'hypothèse de récurrence à a_1. Il existe un contexte de réduction

[9] Comme c'est aussi le résultat de la substitution dans E du terme e à la variable $-$, le lemme des substitutions peut s'appliquer.

E_1 et un radical r_1 tels que $a_1 = E_1[r_1]$ et vérifiant les autres propriétés. On vérifie facilement que $E = E_1\, a_2$ et $r = r_1$ conviennent.

• Autres cas

Ils sont analogues au précédent.

⌞

Contrairement au λ-calcul simple, ce ne sont pas des programmes qui se réduisent, mais des configurations : une configuration décrit l'état courant du programme s'exécutant par un terme clos et par l'état de la mémoire. Le terme clos est le terme de contrôle, qui définit le radical à réduire et la continuation, alors que l'état-mémoire donne le contenu de chacune des références créées. Précisément, un *état-mémoire* s est une fonction de l'ensemble des références dans l'ensemble des valeurs telle que :

- la famille $(l)_{l^A \in \mathrm{dom}\, s}$ est injective, autrement dit les références du domaine de s peuvent être identifiées par leur identificateur,
- l'ensemble des identificateurs $\{l \mid \exists\, A\,.\, l^A \in \mathrm{dom}\, s\}$ est un segment initial de l'ensemble des identificateurs de références, supposé isomorphe à l'ensemble des entiers naturels, et donc bien ordonné,
- pour toute référence l^A du domaine de s, la valeur $s(l^A)$ a pour type A.

Les contraintes imposées au domaine d'un état-mémoire permettent de définir pour tout type A une fonction de *création* de références, ν_A, qui associe à tout état-mémoire s la référence l^A, où l est le minimum du complémentaire de l'ensemble d'identificateurs $\{l \mid \exists\, A\,.\, l^A \in \mathrm{dom}\, s\}$.

Autre opération utile, la *mise à jour* d'un état-mémoire, qui modifie le contenu d'une référence existante, ou ajoute une nouvelle association entre une nouvelle référence et une valeur. À tout état-mémoire s, toute référence l^A et toute valeur v, tels que l^A appartient à $\mathrm{dom}\, s \cup \{\nu_A(s)\}$ et v a pour type A, associons un nouvel état-mémoire, $(s, l^A \mapsto v)$, défini par :

$$(s, l^A \mapsto v)(k^B) \stackrel{def}{=} \begin{cases} s(k^B) & \text{si } k^B \neq l^A\,, \\ v & \text{sinon.} \end{cases}$$

On vérifie facilement que $(s, l^A \mapsto v)$ est bien un état-mémoire.

Une configuration est composée d'un état-mémoire et d'un terme clos typé. Comme toute référence utilisée doit posséder un contenu, on impose que le domaine de l'état-mémoire contient

- les références ayant une occurrence dans le terme de contrôle,
- les références ayant une occurrence dans une valeur référencée par l'état-mémoire.

$$\frac{\emptyset}{(s, (\lambda x : A.\, b)\, v) \to (s, b[v/x])}\ [\beta]$$

$$\frac{\emptyset}{(s, \mathrm{ref}(v)) \to ((s, l^A \mapsto v), l^A)}\ [\mathrm{REF}] \quad (l^A = \nu_A(s))$$

$$\frac{\emptyset}{(s, \mathrm{get}(l^A)) \to (s, s(l^A))}\ [\mathrm{REF\text{-}!}]$$

$$\frac{\emptyset}{(s, \mathrm{set}(l^A, v)) \to ((s, l^A \mapsto v), \mathrm{unit})}\ [\mathrm{REF\text{-}?}]$$

$$\frac{(s, r) \to (s', r')}{(s, E[r]) \to (s', E[r'])}\ [\mathrm{RED}] \quad \left(\begin{array}{l} r \text{ radical} \\ E \neq - \end{array} \right)$$

Tab. 3.7 – Sémantique opérationnelle du langage avec références - La relation de réduction

Il est utile d'étendre aux couples formés d'un état-mémoire et d'un terme l'application $\mathbf{Ref}(-)$ donnant l'ensemble des références ayant une occurrence dans un terme :

$$\mathbf{Ref}(s, e) \;=\; \Big(\bigcup_{l^A \in \mathrm{dom}\, s} \mathbf{Ref}(s(l^A)) \Big) \cup \mathbf{Ref}(e)\,.$$

Ainsi, un couple (s, e) composé d'un état-mémoire s et d'un terme clos typé e forme une *configuration* s'il vérifie la condition suivante :

$$\mathbf{Ref}(s, e) \subseteq \mathrm{dom}\, s\,.$$

De cette condition, on déduit que les références d'une configuration peuvent être identifiées par leur identificateur.
La relation de réduction est définie par un système d'inférence, présenté dans la table 3.7 (p. 231). Un jugement y est de la forme $(s, a) \to (s', a')$, où (s, a) est une configuration, et signifie que (s, a) se réduit en (s', a'). La relation de réduction est bien définie entre configurations et préserve le type du terme de contrôle.

3.2.4 Proposition (Réduction du sujet)
Soit (s, a) une configuration.
Si (s, a) se réduit en (s', a'), alors (s', a') est une configuration et a' possède le même type que a.

Démonstration
On procède par récurrence structurelle sur la preuve de la réduction $(s, a) \to (s', a')$.
• $[\beta]$
a est égal à $(\lambda x : A.\, b)\, v$, et la réduction est $(s, (\lambda x : A.\, b)\, v) \to (s, b[v/x])$. Puisque s est un état-mémoire, le couple $(s, b[v/x])$ est une configuration si et seulement si $b[v/x]$ est typé. Par le lemme des substitutions, $b[v/x]$ est typé, de même type que b dans l'environnement $(x : A)$. Il en résulte que $b[v/x]$ a la même type que a.
• [REF], [REF-!], [REF-?]
C'est sans aucune difficulté.
• [RED]
a est égal à $E[r]$ et la réduction est $(s, E[r]) \to (s', E[r'])$. Elle se déduit de la réduction $(s, r) \to (s', r')$. De l'hypothèse de récurrence, on déduit que (s', r') est une configuration et que r' a même type que r. Il s'ensuit que $E[r']$ a le même type que $E[r]$ et que $(s', E[r'])$ est une configuration.
⌞

La relation de réduction est aussi déterministe et totale, au sens donné par la proposition suivante.

3.2.5 Proposition (Réduction : déterminisme et totalité)
Soit (s, a) une configuration.
Si a est une valeur, alors (s, a) ne se réduit pas.
Sinon, (s, a) se réduit en une unique configuration.

Démonstration
La relation de réduction concerne la réduction de configurations (s, a) où a se décompose en un radical dans un contexte de réduction. Par le lemme de la décomposition, a n'est pas une valeur.
Soit (s, a) une configuration telle que a n'est pas une valeur.
L'existence d'une configuration (s', a') telle que (s, a) se réduit en (s', a') découle de la décomposition de a en un radical dans un contexte de réduction, et de la réduction de toute configuration (s, r), où r est un radical.
L'unicité se démontre par récurrence sur la preuve de $(s, a) \to (s', a')$ et découle de l'unicité de la décomposition en un radical dans un contexte de

$$
\begin{array}{rcll}
m & ::= & (A, \iota) & \text{(type et information)} \\
\\
e & ::= & x & \text{(variable)} \\
 & | & \lambda x^m\, e & \text{(abstraction)} \\
 & | & @^m(e, e) & \text{(application)} \\
 & | & \mathrm{unit}^m & \text{(unité)} \\
 & | & l^m & \text{(référence)} \\
 & | & \mathrm{ref}^m(e) & \text{(création de référence)} \\
 & | & \mathrm{get}^m(e) & \text{(lecture)} \\
 & | & \mathrm{set}^m(e, e) & \text{(écriture)} \\
\\
A & ::= & \mathrm{Unit} & \text{(type singleton)} \\
 & | & A \to A & \text{(type fonctionnel)} \\
 & | & \mathrm{Ref}(A) & \text{(type des références)}
\end{array}
$$

TAB. 3.8 – Syntaxe du langage annoté

réduction.
⌞

On peut donc définir sans ambiguïté la *trace* d'un programme a comme la suite maximale de configurations obtenues par réduction à partir de la configuration initiale $(\emptyset, a)$, où $\emptyset$ est l'état-mémoire de domaine vide. Le résultat d'évaluation est soit une configuration dont le terme de contrôle est une valeur, soit la divergence.

3.2.2 Annoter les termes

Pour étudier le confinement, nous utilisons un langage annoté, ce qui signifie que chaque symbole d'opération reçoit une étiquette. Une étiquette est un couple, dont la première composante est un type et la seconde une information. Si m est une étiquette, on note $(m.\mathrm{T}, m.\mathrm{I})$ le couple égal à m, composé du type $m.\mathrm{T}$ et de l'information $m.\mathrm{I}$. La syntaxe du langage annoté est présentée dans la table 3.8. On remarque que si on efface les étiquettes, on retrouve la grammaire du langage non annoté, à deux exceptions près, l'abstraction et la référence. Dans ces deux cas, on utilise l'étiquette pour

obtenir le type. Du fait de cette similitude, nous omettons par la suite certains détails déjà décrits pour le langage non annoté et mettons l'accent sur les quelques différences. Le système de types et la sémantique par réduction du langage annoté sont calqués sur ceux du langage non annoté ; seule différence, les opérateurs étiquetés portent avec eux leur type et une information, que les réductions préservent.
On impose la même condition de bonne formation aux termes étiquetés : pour tout terme e, il est supposé que la famille $(l)_{l^m \in \mathbf{Ref}(e)}$ est injective, autrement dit il est supposé que les références d'un terme peuvent être identifiées par leur identificateur.
Il nous arrivera de décrire la grammaire engendrant les termes sous une forme abrégée, de la manière suivante :

$$e \quad ::= \quad \ldots \mid f_i^m(\underbrace{e, \ldots, e}_{\text{i fois}}) \,,$$

les points de suspension correspondant aux règles détaillées et la dernière règle étant générique, f_i^m représentant tout symbole fonctionnel d'arité i ($i \in \mathbb{N}$) d'étiquette m.
Par exemple, l'étiquette d'un terme e, notée $e.\mathrm{L}$, peut se définir simplement ainsi :

$$\begin{aligned} x.\mathrm{L} &\stackrel{def}{=} \bot \,, \\ f_i^m(e_i)_i.\mathrm{L} &\stackrel{def}{=} m \,. \end{aligned}$$

Le système de types est calqué sur le précédent et est conçu de manière à assurer que l'étiquette d'un terme indique son type. Il est décrit dans la table 3.9 (p. 235) ; il permet de définir inductivement l'ensemble des jugements de typage valides.
Une substitution est définie comme précédemment. La sémantique opérationnelle du langage annoté s'inspire de celle qui précède. Comme toujours, il est indispensable de pouvoir décomposer un terme clos en ou bien une valeur, ou bien un radical dans un contexte de réduction, cette dernière décomposition étant unique.
Une *valeur* est ou bien une abstraction close, ou bien la valeur unité, ou bien une référence, ce qu'on résume ainsi :

$$v \quad ::= \quad \lambda x^m \, e \mid \mathrm{unit}^m \mid l^m \,,$$

où $\mathbf{FV}(\lambda x^m \, e) = \emptyset$.
Un *radical* est ou bien l'application d'une abstraction close à une valeur, ou

$$\frac{\emptyset}{\Gamma \vdash x \,:\, \Gamma(x)}\,[\text{intro-Var}] \quad (x \in \operatorname{dom}\Gamma)$$

$$\frac{\Gamma.(x \,:\, A) \vdash e \,:\, B}{\Gamma \vdash \lambda x^m\, e \,:\, A \to B}\,[\text{intro-}\to] \quad (A \to B = m.\mathrm{T})$$

$$\frac{\Gamma \vdash f \,:\, A \to B \quad \Gamma \vdash a \,:\, A}{\Gamma \vdash @^m(f,a) \,:\, B}\,[\text{élim-}\to] \quad (B = m.\mathrm{T})$$

$$\frac{\emptyset}{\Gamma \vdash \mathrm{unit}^m \,:\, \mathrm{Unit}}\,[\text{intro-Unit}] \quad (\mathrm{Unit} = m.\mathrm{T})$$

$$\frac{\emptyset}{\Gamma \vdash l^m \,:\, \mathrm{Ref}(A)}\,[\text{intro-Ref/loc}] \quad (\mathrm{Ref}(A) = m.\mathrm{T})$$

$$\frac{\Gamma \vdash e \,:\, A}{\Gamma \vdash \mathrm{ref}^m(e) \,:\, \mathrm{Ref}(A)}\,[\text{intro-Ref/new}] \quad (\mathrm{Ref}(A) = m.\mathrm{T})$$

$$\frac{\Gamma \vdash e \,:\, \mathrm{Ref}(A)}{\Gamma \vdash \mathrm{get}^m(e) \,:\, A}\,[\text{élim-Ref/ !}] \quad (A = m.\mathrm{T})$$

$$\frac{\Gamma \vdash e_1 \,:\, \mathrm{Ref}(A) \quad \Gamma \vdash e_2 \,:\, A}{\Gamma \vdash \mathrm{set}^m(e_1, e_2) \,:\, \mathrm{Unit}}\,[\text{élim-Ref/ ?}] \quad (\mathrm{Unit} = m.\mathrm{T})$$

Tab. 3.9 – Système de types du langage annoté

bien la création d'une référence, ou bien la lecture d'une référence, ou bien l'écriture d'une référence, ce qui donne :

$$r \quad ::= \quad @^n(\lambda x^m\, e, v) \mid \mathrm{ref}^m(v) \mid \mathrm{get}^m(l^n) \mid \mathrm{set}^m(l^n, v)\,.$$

Un *contexte de réduction* se construit autour de la place $-$ de la manière suivante :

$$\begin{array}{rcl} E & ::= & - \\ & \mid & @^n(v, E) \mid @^n(E, e) \\ & \mid & \mathrm{ref}^m(E) \mid \mathrm{get}^m(E) \mid \mathrm{set}^m(E, e) \mid \mathrm{set}^m(v, E)\,, \end{array}$$

où $\mathbf{FV}(e) = \emptyset$.
Un *état-mémoire* s est une fonction de l'ensemble des références dans l'ensemble des valeurs telle que :

- la famille $(l)_{l^m \in \mathrm{dom}\, s}$ est injective, autrement dit les références du domaine de s peuvent être identifiées par leur identificateur,
- l'ensemble des identificateurs $\{l \mid \exists\, m\,.\, l^m \in \mathrm{dom}\, s\}$ est un segment initial de l'ensemble des identificateurs de références, supposé isomorphe à l'ensemble des entiers naturels, et donc bien ordonné,
- pour toute référence l^m du domaine de s, la valeur $s(l^m)$ a pour type A, où A vérifie l'égalité $\mathrm{Ref}(A) = m.\mathrm{T}$.

Étant donné une étiquette m, dont le type $m.\mathrm{T}$ est un type de références, la fonction de *création* de références, ν_m, associe à tout état-mémoire s la référence l^m, où l est le minimum du complémentaire de l'ensemble d'identificateurs $\{l \mid \exists\, m\,.\, l^m \in \mathrm{dom}\, s\}$. Quant à la *mise à jour* d'un état-mémoire, elle associe à tout état-mémoire s, toute référence l^m et toute valeur v, tels que l^m appartient à $\mathrm{dom}\, s \cup \{\nu_m(s)\}$ et v a pour type A vérifiant $\mathrm{Ref}(A) = m.\mathrm{T}$, un nouvel état-mémoire, $(s, l^m \mapsto v)$, défini par :

$$(s, l^m \mapsto v)(k^n) \stackrel{def}{=} \begin{cases} s(k^n) & \text{si } k^n \neq l^m\,, \\ v & \text{sinon.} \end{cases}$$

Une configuration est définie comme précédemment : un couple (s, e) composé d'un état-mémoire s et d'un terme clos typé e forme une *configuration* s'il vérifie la condition suivante :

$$\mathbf{Ref}(s, e) \subseteq \mathrm{dom}\, s\,.$$

La relation de réduction est définie par le système d'inférence présenté dans la table 3.10 (p. 237). On vérifie facilement comme précédemment que la

$$\frac{\emptyset}{(s, @^m(\lambda x^n\, e, v)) \to (s, e[v/x])}\,[\beta]$$

$$\frac{\emptyset}{(s, \mathrm{ref}^m(v)) \to ((s, l^m \mapsto v), l^m)}\,[\mathrm{REF}] \quad (l^m = \nu_m(s))$$

$$\frac{\emptyset}{(s, \mathrm{get}^m(l^n)) \to (s, s(l^n))}\,[\text{REF-}!]$$

$$\frac{\emptyset}{(s, \mathrm{set}^m(l^n, v)) \to ((s, l^n \mapsto v), \mathrm{unit}^m)}\,[\text{REF-}?]$$

$$\frac{(s, r) \to (s', r')}{(s, E[r]) \to (s', E[r'])}\,[\mathrm{RED}] \quad \left(\begin{array}{l} r \text{ radical} \\ E \neq - \end{array}\right)$$

TAB. 3.10 – Sémantique opérationnelle du langage annoté - Relation de réduction

relation de réduction est bien définie entre configurations, préserve le typage et le type du terme de contrôle par réduction du sujet, enfin est déterministe et totale.

Pour terminer, précisons formellement le rapport entre le langage annoté et le langage non annoté.
La fonction d'*effacement* des étiquettes, notée $\downarrow$, est définie inductivement ainsi :

$$\begin{array}{rcl} \downarrow(x) & = & x\,, \\ \downarrow(\lambda x^m\, e) & = & \lambda x : A.\downarrow(e) \quad (m.\mathrm{T} = A \to B)\,, \\ \downarrow(l^m) & = & l^A \quad (m.\mathrm{T} = \mathrm{Ref}(A))\,, \\ \downarrow(f_i^m(e_j)_j) & = & f_i(\downarrow(e_j))_j\,. \end{array}$$

Elle conserve la relation de typage.

3.2.6 Proposition (Effacement : Conservation du typage)
Si le terme étiqueté e a pour type A dans l'environnement Γ, alors $\downarrow(e)$ a pour type A dans l'environnement Γ.
Réciproquement, si le terme non étiqueté a a pour type A dans l'environne-

ment Γ*, il existe un terme étiqueté* e *de type* A *dans l'environnement* Γ *tel que* $\downarrow(e) = a$*.*

Démonstration
La proposition se montre facilement par récurrence structurelle sur les preuves de $\Gamma \vdash e : A$ et de $\Gamma \vdash a : A$.
⌞

La conservation de la relation de typage permet d'associer à tout terme non étiqueté typé un terme étiqueté de même type, antécédent par la fonction d'effacement. Un autre antécédent de même type ne diffère que par l'information qu'il porte, comme le montre la proposition suivante.

3.2.7 Proposition (**Effacement : Représentation de l'équivalence canonique** **)**

Considérons la relation d'équivalence $\sim$*, reliant deux termes étiquetés s'ils sont égaux à condition de restreindre la comparaison des étiquettes aux types, autrement dit considérons la relation* $\sim$ *engendrée inductivement par le système d'inférence formé des règles suivantes :*

$$\frac{\emptyset}{(x, x)},$$

$$\frac{(e_j^1, e_j^2)_j}{(f^m(e_j^1)_j, f^n(e_j^2)_j)} \quad (m.\mathrm{T} = n.\mathrm{T})\,.$$

Alors, pour tous termes étiquetés e_1 *et* e_2 *typés dans un même environnement de typage, nous avons :*

$$e_1 \sim e_2 \Leftrightarrow \downarrow(e_1) = \downarrow(e_2)\,.$$

Démonstration
• $e_1 \sim e_2 \Rightarrow \downarrow(e_1) = \downarrow(e_2)$
On procède par récurrence structurelle sur la preuve de $e_1 \sim e_2$. C'est immédiat.
• $\downarrow(e_1) = \downarrow(e_2) \Rightarrow e_1 \sim e_2$
On procède par récurrence structurelle sur la preuve de typage de e_1 dans Γ. Examinons les différents cas pour la règle d'inférence concluant la preuve. On suppose à chaque fois que e_2 est un terme typé dans Γ tel que $\downarrow(e_1) = \downarrow(e_2)$, et on cherche à montrer que $e_1 \sim e_2$.
◦ [intro-Var]

Dans ce cas, il existe une variable x telle que $e_1 = e_2 = x$. On peut conclure.
∘ [intro- $\rightarrow$]
La preuve se termine par la règle

$$\frac{\Gamma.(x\ :\ A) \vdash b_1\ :\ B_1}{\Gamma \vdash \lambda x^m\, b_1\ :\ A \rightarrow B_1},$$

où $m.\mathrm{T} = A \rightarrow B_1$.
On déduit de $\downarrow (e_1) = \downarrow (e_2)$ et du typage de e_2 dans Γ l'existence d'une étiquette n et d'un terme b_2 tels que $e_2 = \lambda x^n\, b_2$, $\downarrow (b_1) = \downarrow (b_2)$, $n.\mathrm{T} = A \rightarrow B_2$ et $\Gamma.(x\ :\ A) \vdash b_2\ :\ B_2$.
De l'hypothèse de récurrence appliquée à $\Gamma.(x\ :\ A) \vdash b_1\ :\ B_1$, on obtient que $b_1 \sim b_2$.
Si b_1 est une variable, alors $b_1 = b_2$, puis $B_1 = B_2$, soit $m.\mathrm{T} = n.\mathrm{T}$.
Sinon, par définition de $\sim$, on a $b_1.\mathrm{L.T} = b_2.\mathrm{L.T}$, soit $B_1 = B_2$, soit $m.\mathrm{T} = n.\mathrm{T}$.
Finalement, on a bien $e_1 \sim e_2$.
∘ Autres cas
Le raisonnement est analogue au précédent.
⌞

Ainsi, il est possible d'associer à tout terme clos non étiqueté a de type A, et à toute information ι, l'unique terme clos étiqueté, noté $\langle a \rangle^\iota$, de type A, vérifiant $\downarrow (\langle a \rangle^\iota) = a$ et entièrement étiqueté par ι, coloré en ι pourrait-on dire : toute étiquette m ayant une occurrence dans $\langle a \rangle^\iota$ vérifie $m.\mathrm{I} = \iota$.
La fonction d'effacement peut être étendue aux états-mémoire de la manière suivante. Si s est un état-mémoire étiqueté, comme la famille $(l)_{l^m \in \mathrm{dom}\, s}$ est injective, on peut associer à tout identificateur de référence l appartenant à l'image de cette famille l'étiquette m_l telle que l^{m_l} appartient au domaine de s ; $\downarrow (s)$ est alors la fonction décrite par la famille $\big(\downarrow (s(l^{m_l})) \big)_{l^A \in \downarrow(\mathrm{dom}\, s)}$.
Finalement, la fonction d'effacement peut être étendue aux configurations de la manière suivante : si (s, e) est une configuration étiquetée, alors $\downarrow (s, e)$ est égal à $(\downarrow (s), \downarrow (e))$. Par préservation de la relation de typage, c'est bien une configuration. La réduction est aussi conservée par effacement.

3.2.8 Proposition (Effacement : Conservation de la réduction)
Si (s, e) est une configuration étiquetée se réduisant en (s', e'), alors $\downarrow (s, e)$ se réduit en $\downarrow (s', e')$; si (s, e) est une configuration étiquetée ne se réduisant pas, alors $\downarrow (s, e)$ ne se réduit pas.
Réciproquement, si (t, a) est une configuration non étiquetée se réduisant en (t', a'), alors il existe deux configurations étiquetées (s, e) et (s', e') telles

que (s, e) *se réduit en* (s', e')*, et* $\downarrow (s, e) = (t, a)$ *et* $\downarrow (s', e') = (t,' a')$ *; si* (t, a) *est une configuration non étiquetée ne se réduisant pas, alors il existe une configuration étiquetée* (s, e) *telle que* (s, e) *ne se réduit pas et* $\downarrow (s, e) = (t, a)$*.*

Démonstration
Pour la première partie de la proposition, concernant la préservation de la réduction, il suffit de remarquer tout d'abord que la fonction d'effacement $\downarrow$ transforme une valeur en une valeur, un radical en un radical, un contexte de réduction en un contexte de réduction. En remarquant que la fonction d'effacement commute avec les substitutions, on constate alors qu'un radical dans un contexte de réduction est transformé en un radical dans un contexte de réduction, et on peut procéder par récurrence sur la preuve de la réduction pour conclure.
Pour la seconde partie, considérons une configuration non étiquetée (t, a). Soit ι une information quelconque. Définissons s ainsi, pour toute référence l^A du domaine de t :

$$s(l^{(\mathrm{Ref}(A),\iota)}) \stackrel{def}{=} \langle t(l^A)\rangle^\iota \,.$$

Il est clair que $(s, \langle a\rangle^\iota)$ est une configuration étiquetée telle que $\downarrow (s, \langle a\rangle^\iota) = (t, a)$. On applique la première partie de la proposition pour conclure.
∟

3.2.3 Les types à la frontière

Pour étudier le confinement, nous utilisons le langage annoté que nous venons de décrire, en donnant un sens particulier aux informations composant les étiquettes : elles représentent l'origine des termes. Comme un terme peut provenir ou bien du code mobile ou bien du code local, l'ensemble des informations est supposé égal à $\{\mathsf{mo}, \mathsf{lo}\}$: l'information mo indique comme provenance le code mobile, l'information lo le code local. Si ι appartient à $\{\mathsf{mo}, \mathsf{lo}\}$, on note $\bar{\iota}$ l'autre élément de $\{\mathsf{mo}, \mathsf{lo}\}$.

Nous définissons maintenant la *frontière* entre le code mobile et le code local au sein d'un terme. Considérons un terme e, et un de ses sous-termes $f^m(\ldots, d, \ldots)$, où d a pour étiquette n ; le sous-terme d est à la frontière dans e si son origine est différente de celle de l'opérateur f, soit si

$$n.\mathrm{I} \neq m.\mathrm{I}\,.$$

Il existe donc deux sortes de frontières, l'une, dite *sortante*[10], correspondant

[10]Le code local placé à cette frontière peut sortir de l'environnement, le code mobile opérant alors sur lui.

au cas où d a pour origine le code local, soit $n.\mathrm{I} = \mathsf{lo}$, l'autre, dite *entrante*[11], correspondant au cas où d a pour origine le code mobile, soit $n.\mathrm{I} = \mathsf{mo}$.
Plutôt que de déterminer les termes apparaissant à la frontière, nous réalisons une approximation, en nous intéressant seulement à leur type. Définissons donc deux ensembles, formés des types apparaissant aux frontières sortante et entrante respectivement au sein d'un terme.
Soit ι un élément de $\{\mathsf{mo}, \mathsf{lo}\}$. Si m est une étiquette, on définit une application F^m_ι, associant à chaque terme étiqueté e

– le singleton $\{e.\mathrm{L.T}\}$, si e et m portent les informations ι et $\bar{\iota}$ respectivement :

$$\begin{aligned} e.\mathrm{L.I} &= \iota\,, \\ m.\mathrm{I} &= \bar{\iota}\,, \end{aligned}$$

– l'ensemble vide sinon.

Autrement dit, l'application F^m_ι appliquée à un terme e détermine si e est à la frontière, sortante ou entrante suivant la valeur de ι, lorsqu'il est placé sous un opérateur étiqueté par m, et le cas échéant, renvoie son type. Ainsi, pour toute variable x, $\mathrm{F}^m_\iota(x) = \emptyset$; pour un terme e d'étiquette (A, ι), $\mathrm{F}^m_\iota(e)$ est égal à $\{A\}$ si $m.\mathrm{I} = \bar{\iota}$, à l'ensemble vide sinon, et $\mathrm{F}^m_{\bar{\iota}}(e) = \emptyset$.
À tout terme étiqueté e, on associe l'ensemble noté $\mathcal{F}_\iota(e)$ des types appartenant à la frontière, sortante si $\iota = \mathsf{lo}$, entrante si $\iota = \mathsf{mo}$, défini récursivement par les équations suivantes :

$$\begin{aligned} \mathcal{F}_\iota(x) &= \emptyset\,, \\ \mathcal{F}_\iota(f^m_j(e_1, \ldots, e_j)) &= \bigcup_{1 \le k \le j} \mathcal{F}_\iota(e_k) \cup \mathrm{F}^m_\iota(e_k)\,. \end{aligned}$$

Ainsi, dans un terme, un type appartient à la frontière sortante si c'est le type d'un sous-terme d'origine lo, placé immédiatement sous un sous-terme d'origine mo, et inversement pour la frontière entrante. Par exemple, pour le terme étiqueté e égal à

$$@^{(B,\mathsf{lo})}(f^{(A\to B,\mathsf{mo})}, a^{(A,\mathsf{lo})})\,,$$

$\mathcal{F}_{\mathsf{mo}}(e)$ est égal à

$$\{A \to B\} \cup \mathcal{F}_{\mathsf{mo}}(f) \cup \mathcal{F}_{\mathsf{mo}}(a)\,.$$

L'application est étendue aux états-mémoire de la manière suivante :

$$\mathcal{F}_\iota(s) = \bigcup_{l^m \in \mathrm{dom}\, s} \mathcal{F}_\iota(s(l^m)) \cup \mathrm{F}^m_\iota(s(l^m))\,.$$

[11] Le code mobile placé à cette frontière peut entrer dans l'environnement qui opère alors sur lui.

puis aux configurations ainsi :

$$\mathcal{F}_\iota(s,e) = \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(e)\,.$$

Ainsi, dans une configuration (s,e), un type appartient à la frontière sortante
- ou s'il appartient à la frontière sortante dans e,
- ou s'il appartient à la frontière sortante dans une valeur $s(l^m)$,
- ou s'il est le type d'une valeur $s(l^m)$ d'origine lo, l^m ayant pour origine mo,

et de même pour la frontière entrante, en permutant les origines.
Pour simplifier, on définit la frontière d'un terme comme le couple

$$(\mathcal{F}_{\mathsf{lo}}(e), \mathcal{F}_{\mathsf{mo}}(e))\,,$$

de même la frontière d'une configuration (s,e) comme le couple

$$(\mathcal{F}_{\mathsf{lo}}(s,e), \mathcal{F}_{\mathsf{mo}}(s,e))\,.$$

Plus généralement, on appelle frontière tout couple formé de deux ensembles de types.

Notre premier objectif est de déterminer à partir d'un programme un majorant de l'ensemble des frontières apparaissant dans sa trace d'exécution, le majorant étant calculé à partir de la frontière de la configuration initiale. Il doit donc être possible de déterminer la manière dont évolue la frontière lors d'une réduction. Une hypothèse portant sur les variables liées est alors nécessaire, comme le montre l'exemple suivant.
Considérons deux valeurs $\langle f\rangle^{\mathsf{mo}}$ et $\langle v\rangle^{\mathsf{lo}}$, formées de code d'origine mo et lo respectivement, et de types respectifs $A \to B$ et A. Le programme

$$@^{(B,\mathsf{lo})}(\lambda x^{(A\to B,\mathsf{lo})}\, @^{(B,\mathsf{mo})}(\langle f\rangle^{\mathsf{mo}}, x), \langle v\rangle^{\mathsf{lo}})$$

a pour frontière $(\emptyset, \{B\})$. Il se réduit en $@^{(B,\mathsf{mo})}(\langle f\rangle^{\mathsf{mo}}, \langle v\rangle^{\mathsf{lo}})$, qui a pour frontière $(\{A\}, \emptyset)$. Comme A et B peuvent être quelconques, une hypothèse concernant l'étiquetage s'avère nécessaire si l'on veut rapporter la frontière après réduction à la frontière avant : elle assure qu'on ne traverse aucune frontière, d'une part, entre une variable liée et son lieur, d'autre part, entre une variable libre et la racine du terme. C'est l'hypothèse de cohérence des origines.

3.2.9 Définition (Cohérence des origines)
*Un terme étiqueté e est d'*origine cohérente *si l'une des conditions suivantes est vérifiée :*

$$\frac{\emptyset}{x} \quad (x \text{ variable})$$

$$\frac{e_1 \dots e_j}{f_j^m(e_1, \dots, e_j)} \quad \left(\begin{array}{l} \forall\, k \in \{1, \dots, j\}\,. \\ \quad \mathbf{FV}(e_k) \neq \emptyset \\ \quad \Rightarrow \mathrm{F}^m_{\mathsf{lo}}(e_k) = \mathrm{F}^m_{\mathsf{mo}}(e_k) = \emptyset \end{array} \right)$$

TAB. 3.11 – Termes étiquetés : Cohérence des origines

(i) *e est une variable,*

(ii) *e est égal à* $f_j^m(e_1, \dots, e_j)$*, et pour tout entier* k *compris entre* 1 *et* j*,*
- e_k *est d'origine cohérente,*
- *si* e_k *possède une variable libre,* e_k *n'appartient pas à la frontière de* e*.*

Formellement, un terme étiqueté e est d'origine cohérente si e appartient à l'ensemble engendré inductivement par le système d'inférence de la table 3.11 (p. 243).
Une configuration (s, e) *est d'origine cohérente si*
- *le terme de contrôle e est d'origine cohérente,*
- *pour toute référence* l^m *du domaine de s, la valeur* $s(l^m)$ *est d'origine cohérente.*

Ainsi, dans l'exemple précédent, le terme

$$\lambda x^{(A \to B, \mathsf{lo})}\, @^{(B, \mathsf{mo})}(\langle f \rangle^{\mathsf{mo}}, x)$$

n'est pas d'origine cohérente. En revanche, les termes

$$\lambda x^{(A \to B, \mathsf{mo})}\, @^{(B, \mathsf{mo})}(\langle f \rangle^{\mathsf{mo}}, x)$$

et

$$\lambda x^{(A \to B, \mathsf{lo})}\, @^{(B, \mathsf{lo})}(\langle f \rangle^{\mathsf{mo}}, x)$$

le sont. Pour ces deux cas, voyons comment évolue la frontière lors de la réduction du programme corrigé. Les programmes

$$@^{(B, \mathsf{lo})}(\lambda x^{(A \to B, \mathsf{mo})}\, @^{(B, \mathsf{mo})}(\langle f \rangle^{\mathsf{mo}}, x), \langle v \rangle^{\mathsf{lo}})$$

et

$$@^{(B, \mathsf{lo})}(\lambda x^{(A \to B, \mathsf{lo})}\, @^{(B, \mathsf{lo})}(\langle f \rangle^{\mathsf{mo}}, x), \langle v \rangle^{\mathsf{lo}})$$

ont pour frontière $(\emptyset, \{A \to B\})$. Le premier se réduit en $@^{(B,\mathsf{mo})}(\langle f\rangle^{\mathsf{mo}}, \langle v\rangle^{\mathsf{lo}})$, de frontière $(\{A\}, \emptyset)$, le second en $@^{(B,\mathsf{lo})}(\langle f\rangle^{\mathsf{mo}}, \langle v\rangle^{\mathsf{lo}})$, de frontière $(\emptyset, \{A \to B\})$. Dans les deux cas, il est possible de relier les types appartenant à la frontière après réduction à ceux qui y appartiennent avant. C'est cette intuition que nous développons et formalisons maintenant.

Désormais, nous allons utiliser des termes d'origine cohérente. Montrons donc que cette propriété est préservée par réduction.
Commençons par étudier la décomposition dans un contexte de réduction. Précisons la définition de la propriété pour un contexte de réduction : un contexte de réduction est d'origine cohérente s'il appartient à l'ensemble engendré inductivement par le système d'inférence de la définition précédente, à condition de ne pas considérer la place comme une variable libre. Par exemple, le contexte $\mathrm{get}^m(\mathrm{get}^n(-))$ est d'origine cohérente, quels que soient m et n. On vérifie aisément que les contextes de réduction d'origine cohérente sont engendrés par la grammaire suivante :

$$\begin{array}{rcl} E & ::= & - \\ & | & @^n(E, e) \mid @^n(v, E) \\ & | & \mathrm{ref}^m(E) \mid \mathrm{get}^m(E) \mid \mathrm{set}^m(E, e) \mid \mathrm{set}^m(v, E)\,, \end{array}$$

où la valeur v et le terme clos e sont d'origine cohérente.
Voyons le premier lemme, utilisé pour la règle d'inférence [RED].

3.2.10 Lemme
Soient E un contexte de réduction et e un terme clos.
Si $E[e]$ est d'origine cohérente, alors E et e le sont.

Démonstration
C'est immédiat, par récurrence structurelle sur E.
⌞

Intéressons-nous maintenant aux substitutions, utilisées dans la β-réduction ; si on remplace une variable libre par un terme clos, la cohérence est préservée.

3.2.11 Lemme
Soient e et a deux termes étiquetés.
Supposons que a soit clos, et que e et a soient d'origine cohérente. Alors $e[a/x]$ est d'origine cohérente.

Démonstration
Soit a un terme clos d'origine cohérente. On montre par récurrence structurelle sur e que si e est d'origine cohérente, alors $e[a/x]$ aussi. On traite le

seul cas intéressant où $e = f_j^m(e_1, \ldots, e_j)$. On suppose que e est d'origine cohérente.
On doit montrer que pour tout entier k compris entre 1 et j, on a :
– $e_k[a/x]$ est d'origine cohérente,
– si $\mathbf{FV}(e_k[a/x]) \neq \emptyset$, alors $\mathrm{F}^m_{\mathsf{lo}}(e_k[a/x]) = \mathrm{F}^m_{\mathsf{mo}}(e_k[a/x]) = \emptyset$.
Soit $k \in \{1, \ldots, j\}$.
La première condition découle de l'hypothèse de récurrence, puisque e_k est d'origine cohérente.
Pour la seconde, examinons les différents cas possibles pour e_k.
Si $e_k = x$, alors $e_k[a/x] = a$ et par hypothèse, $\mathbf{FV}(e_k[a/x]) = \emptyset$.
Si e_k est égal à une variable y différente de x, alors $e_k[a/x] = y$ et par définition $\mathrm{F}^m_{\mathsf{lo}}(y) = \mathrm{F}^m_{\mathsf{mo}}(y) = \emptyset$.
Sinon, $e_k = g^n(\ldots)$, donc $e_k[a/x] = g^n(\ldots)$.
On a alors les égalités $\mathrm{F}^m_{\mathsf{lo}}(e_k[a/x]) = \mathrm{F}^m_{\mathsf{lo}}(e_k)$ et $\mathrm{F}^m_{\mathsf{mo}}(e_k[a/x]) = \mathrm{F}^m_{\mathsf{mo}}(e_k)$.
Comme e_k est d'origine cohérente et comme on a l'inclusion $\mathbf{FV}(e_k[a/x]) \subseteq \mathbf{FV}(e_k)$, puisque a est clos, il vient que si $\mathbf{FV}(e_k[a/x]) \neq \emptyset$, alors on a $\mathrm{F}^m_{\mathsf{lo}}(e_k[a/x]) = \mathrm{F}^m_{\mathsf{mo}}(e_k[a/x]) = \emptyset$.
⌞

Nous pouvons maintenant démontrer la préservation de la cohérence par réduction.

3.2.12 Proposition (Cohérence des origines : Préservation par réduction)

Soit t_1 une configuration se réduisant en t_2.
Si t_1 est d'origine cohérente, alors il en est de même de t_2.

Démonstration
Procédons par récurrence sur la preuve de $t_1 \to t_2$.
• $[\beta]$
On utilise le lemme 3.2.11 (p. 244) pour conclure.
• [REF], [REF-!], [REF-?]
C'est immédiat.
• [RED]
t_1 est de la forme $(s, E[r])$, t_2 de la forme $(s', E[r'])$, avec $(s, r) \to (s', r')$.
Si t_1 est d'origine cohérente, en utilisant le lemme 3.2.10 (p. 244), il en est de même pour E et (s, r), puis par hypothèse de récurrence, pour (s', r'), et enfin, par le lemme 3.2.11 (p. 244), il en est de même pour t_2.
⌞

Venons-en à la majoration de la frontière d'une configuration. À tout type C, nous allons associer deux ensembles de types, $\mathcal{T}_+(C)$ et $\mathcal{T}_-(C)$.

$$\frac{\emptyset}{C \vdash C \,:\, +}\,[+]$$

$$\frac{C \vdash A \to B \,:\, *}{C \vdash A \,:\, \overline{*}}\,[- \to]\quad (* \in \{-,+\})$$

$$\frac{C \vdash A \to B \,:\, *}{C \vdash B \,:\, *}\,[\to +]\quad (* \in \{-,+\})$$

$$\frac{C \vdash \mathrm{Ref}(A) \,:\, *}{C \vdash A \,:\, \overline{*}}\,[\mathrm{Ref}(-)]\quad (* \in \{-,+\})$$

$$\frac{C \vdash \mathrm{Ref}(A) \,:\, *}{C \vdash A \,:\, *}\,[\mathrm{Ref}(+)]\quad (* \in \{-,+\})$$

Tab. 3.12 – Types entrants et sortants

L'ensemble $\mathcal{T}_+(C)$ est formé des types ayant une occurrence positive dans C ou étant gardé dans C par le constructeur $\mathrm{Ref}(-)$; on les appelle les *types sortants* de C ; l'ensemble $\mathcal{T}_-(C)$ est formé des types ayant une occurrence négative dans C ou étant gardé dans C par le constructeur $\mathrm{Ref}(-)$; on les appelle les *types entrants* de C. Ces deux ensembles sont définis formellement à partir du système d'inférence de la table 3.12, interprété inductivement. Un jugement est de la forme $C \vdash A \,:\, *$, où $*$ est un élément de la paire $\{-,+\}$; on note $\overline{*}$ l'autre élément de la paire. Précisément, les ensembles $\mathcal{T}_+(C)$ et $\mathcal{T}_-(C)$ s'obtiennent ainsi :

- le type A appartient à $\mathcal{T}_+(C)$ si le jugement $C \vdash A \,:\, +$ est valide dans le système d'inférence,
- le type A appartient à $\mathcal{T}_-(C)$ si le jugement $C \vdash A \,:\, -$ est valide dans le système d'inférence.

On peut étendre les applications $\mathcal{T}_+(-)$ et $\mathcal{T}_-(-)$ aux ensembles de types de la manière suivante : si $\mathcal{C}$ est un ensemble de types, alors

$$\begin{aligned}\mathcal{T}_+(\mathcal{C}) &= \bigcup_{C \in \mathcal{C}} \mathcal{T}_+(C)\,,\\ \mathcal{T}_-(\mathcal{C}) &= \bigcup_{C \in \mathcal{C}} \mathcal{T}_-(C)\,.\end{aligned}$$

Nous allons relier ces ensembles à la notion de frontière de la manière suivante. Étant donné une frontière $(\mathcal{C}_1, \mathcal{C}_2)$ (c'est-à-dire un couple formé de

deux ensembles de types), définissons l'ensemble des types frontaliers par :

$$\mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)} \stackrel{def}{=} \mathcal{T}_+(\mathcal{C}_1) \cup \mathcal{T}_-(\mathcal{C}_2)\,.$$

Le couple $(\mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}, \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)})$ est appelé la *frontière accessible* à partir de la frontière $(\mathcal{C}_1, \mathcal{C}_2)$. Nous pouvons maintenant énoncer le théorème principal.

3.2.13 Théorème (Frontière et accessibilité)
Soit t une trace d'exécution telle que la configuration initiale t_0 soit d'origine cohérente.
Alors, pour toute configuration t_i de la trace t, la frontière de t_i est majorée (au sens de l'inclusion des composantes) par la frontière accessible à partir de la frontière de la configuration initiale t_0 :

$$\begin{aligned} \mathcal{F}_{\mathsf{lo}}(t_i) &\subseteq \mathcal{A}_{(\mathcal{F}_{\mathsf{lo}}(t_0),\mathcal{F}_{\mathsf{mo}}(t_0))}\,, \\ \mathcal{F}_{\mathsf{mo}}(t_i) &\subseteq \mathcal{A}_{(\mathcal{F}_{\mathsf{mo}}(t_0),\mathcal{F}_{\mathsf{lo}}(t_0))}\,. \end{aligned}$$

La démonstration se fait par récurrence sur la position dans la trace. Elle nécessite d'étudier l'évolution de la frontière d'une configuration lors de sa réduction. Quelques lemmes préparatoires sont utiles.
Commençons par calculer la frontière pour un terme clos placé dans un contexte de réduction. On dit qu'un contexte de réduction *réduit sous n* s'il admet pour sous-terme un contexte de réduction de la forme $f^n(\ldots, -, \ldots)$. L'introduction de cette définition se justifie par le lemme suivant.

3.2.14 Lemme (Frontière et réduction)
Soient E un contexte de réduction réduisant sous n et e un terme clos, d'origine ι.
Alors :

$$\begin{aligned} \mathcal{F}_\iota(E[e]) &= \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(e) \cup \mathrm{F}^n_\iota(e)\,, \\ \mathcal{F}_{\bar{\iota}}(E[e]) &= \mathcal{F}_{\bar{\iota}}(E) \cup \mathcal{F}_{\bar{\iota}}(e)\,. \end{aligned}$$

Démonstration
On montre par récurrence structurelle sur E que si E réduit sous n, alors pour tout terme clos e d'origine ι, les deux égalités sont vérifiées.
Soit e un terme clos d'origine ι. On suppose à chaque fois que E réduit sous n.
• $E = -$
C'est trivial.

• $E = @^m(F, a)$
Supposons $\tau \in \{\mathsf{lo}, \mathsf{mo}\}$. On a :

$$\mathcal{F}_\tau(E[e]) \quad = \quad \mathcal{F}_\tau(F[e]) \cup \mathrm{F}^m_\tau(F[e]) \cup \mathcal{F}_\tau(a) \cup \mathrm{F}^m_\tau(a)\,.$$

Envisageons deux cas pour F.
◦ $F = -$
On a $F[e] = e$, $n = m$ et $\mathcal{F}_\tau(E) = \mathcal{F}_\tau(a) \cup \mathrm{F}^n_\tau(a)$, soit finalement

$$\mathcal{F}_\tau(E[e]) \quad = \quad \mathcal{F}_\tau(e) \cup \mathrm{F}^n_\tau(e) \cup \mathcal{F}_\tau(E)\,.$$

On en déduit les deux égalités attendues.
◦ $F \neq -$
E et F réduisent alors sous n. Comme $\mathrm{F}^m_\tau(F[e]) = \mathrm{F}^m_\tau(F)$, il suffit d'appliquer l'hypothèse de récurrence à F pour obtenir les égalités attendues.
• Autres cas
Ils sont analogues au précédent.
⌞

Intéressons-nous maintenant aux effets d'une substitution.

3.2.15 Lemme (Frontière et substitution)
Soient e et a deux termes et x une variable, tels que e soit d'origine cohérente.
Supposons que e soit distinct de x et que x soit libre dans e. Alors, pour tout élément ι de $\{\mathsf{lo}, \mathsf{mo}\}$, on a :

$$\mathcal{F}_\iota(e[a/x]) = \mathcal{F}_\iota(e) \cup \mathcal{F}_\iota(a) \cup \mathrm{F}^{\mathrm{e.L}}_\iota(a)\,.$$

Démonstration
Soit ι un élément de $\{\mathsf{lo}, \mathsf{mo}\}$.
On montre par récurrence sur e que si e d'origine cohérente est distinct de x et x est libre dans e, alors on a l'égalité indiquée.
• $e = x, e = y$
C'est trivialement vérifié.
• $e = f^n_j(e_1, \ldots, e_j)$
On suppose e d'origine cohérente et $x \in \mathbf{FV}(e)$. Par définition, on a :

$$\mathcal{F}_\iota(e[a/x]) = \bigcup_{1 \leq k \leq j} \mathcal{F}_\iota(e_k[a/x]) \cup \mathrm{F}^n_\iota(e_k[a/x])\,.$$

On doit montrer que

$$\mathcal{F}_\iota(e[a/x]) = \big(\bigcup_{1 \leq k \leq j} \mathcal{F}_\iota(e_k) \cup \mathrm{F}^n_\iota(e_k) \big) \cup \mathcal{F}_\iota(a) \cup \mathrm{F}^n_\iota(a)\,.$$

Soit k un entier compris entre 1 et j. Distinguons deux cas suivant que x est une variable libre de e_k ou non.
∘ $x \in \mathbf{FV}(e_k)$
Si $e_k = x$, alors

$$\begin{aligned} \mathcal{F}_\iota(e_k[a/x]) \cup \mathrm{F}^n_\iota(e_k[a/x]) &= \mathcal{F}_\iota(a) \cup \mathrm{F}^n_\iota(a)\,, \\ \mathcal{F}_\iota(e_k) \cup \mathrm{F}^n_\iota(e_k) &= \emptyset\,. \end{aligned}$$

Supposons $e_k \neq x$.
Comme e est d'origine cohérente, e_k l'est aussi et on a $n.\mathrm{I} = e_k.\mathrm{L.I}$. De l'hypothèse de récurrence appliquée à e_k, on obtient

$$\mathcal{F}_\iota(e_k[a/x]) = \mathcal{F}_\iota(e_k) \cup \mathcal{F}_\iota(a) \cup \mathrm{F}^{e_k.\mathrm{L}}_\iota(a)\,.$$

Comme $n.\mathrm{I} = e_k.\mathrm{L.I}$, on a $\mathrm{F}^{e_k.\mathrm{L}}_\iota(a) = \mathrm{F}^n_\iota(a)$. En remarquant que $\mathrm{F}^n_\iota(e_k[a/x]) = \mathrm{F}^n_\iota(e_k)$, on obtient finalement :

$$\mathcal{F}_\iota(e_k[a/x]) \cup \mathrm{F}^n_\iota(e_k[a/x]) = \mathcal{F}_\iota(e_k) \cup \mathrm{F}^n_\iota(e_k) \cup \mathcal{F}_\iota(a) \cup \mathrm{F}^n_\iota(a)\,.$$

∘ $x \notin \mathbf{FV}(e_k)$
Dans ce cas, $e_k[a/x] = e_k$.
Finalement, en réunissant tous les cas, et en remarquant qu'il existe au moins un k tel que $x \in \mathbf{FV}(e_k)$, on obtient :

$$\mathcal{F}_\iota(e[a/x]) = \mathcal{F}_\iota(e) \cup \mathcal{F}_\iota(a) \cup \mathrm{F}^n_\iota(a)\,.$$

⌞

Étudions maintenant l'évolution de la frontière lors de la mise à jour d'un état-mémoire.

3.2.16 Lemme (Frontière et mise à jour de l'état-mémoire)
Soit s un état-mémoire.
Considérons une référence l^n appartenant à l'ensemble $\operatorname{dom} s \cup \{\nu_n(s)\}$. *Alors on a, pour tout élément ι de $\{\mathsf{lo}, \mathsf{mo}\}$:*

$$\mathcal{F}_\iota((s, l^n \mapsto v)) \subseteq \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(v) \cup \mathrm{F}^n_\iota(v)\,.$$

Démonstration
Il suffit de reprendre la définition de la mise à jour et de la frontière.
⌞

Il est désormais possible de décrire l'évolution de la frontière au cours d'une réduction. Pour les majorants, les propriétés de stabilité suivantes nous seront utiles.

3.2.17 Lemme (Propriétés de la frontière accessible)
Soit $(\mathcal{C}_1, \mathcal{C}_2)$ *une frontière. Alors :*

(i) *tout type de* $\mathcal{C}_1$ *appartient à* $\mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}$,

(ii) *si* $A \to B$ *appartient à* $\mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}$, *alors* B *appartient à* $\mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}$,

(iii) *si* $\mathrm{Ref}(A)$ *appartient à* $\mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}$, *alors* A *appartient à* $\mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}$,

(iv) *si* $A \to B$ *appartient à* $\mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}$, *alors* A *appartient à* $\mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)}$,

(v) *si* $\mathrm{Ref}(A)$ *appartient à* $\mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}$, *alors* A *appartient à* $\mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)}$.

Démonstration
Il suffit de reprendre la définition de la frontière accessible et d'appliquer les règles d'inférence du système décrit dans la table 3.12 (p. 246).
⌞

Finalement, nous pouvons rassembler tous ces éléments pour étudier la préservation de la majoration de la frontière par réduction. Considérons une configuration $(s, E[r])$ se réduisant en $(s', E[r'])$. Envisageons deux cas pour le contexte de réduction E.
Si $E = -$, on a, pour tout élément ι de $\{\mathsf{lo}, \mathsf{mo}\}$:

$$\begin{aligned} \mathcal{F}_\iota(s, E[r]) &= \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(r)\,, \\ \mathcal{F}_\iota(s', E[r']) &= \mathcal{F}_\iota(s') \cup \mathcal{F}_\iota(r')\,. \end{aligned}$$

Sinon, il existe une étiquette h telle que E réduit sous h ; dans ce cas, on a :

$$\begin{aligned} \mathcal{F}_\iota(s, E[r]) &= \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(r) \cup \mathrm{F}^h_\iota(r)\,, \\ \mathcal{F}_\iota(s', E[r']) &= \mathcal{F}_\iota(s') \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(r') \cup \mathrm{F}^h_\iota(r')\,. \end{aligned}$$

Remarquons que si $\mathrm{F}^h_\iota(r) \neq \emptyset$, alors $\mathrm{F}^h_\iota(r') \subseteq \mathrm{F}^h_\iota(r)$, puisque r et r' ont même type. Le seul cas posant problème est donc lorsque $\mathrm{F}^h_\iota(r) = \emptyset$ et $\mathrm{F}^h_\iota(r') \neq \emptyset$, ce qui entraîne que $\mathrm{F}^h_{\bar{\iota}}(r) = \mathrm{F}^h_{\bar{\iota}}(r') = \emptyset$: on dit que le contexte de réduction crée une nouvelle frontière avec le réduit r'. Dans notre premier lemme qui concerne ce cas, nous allons montrer que si le type de la frontière nouvellement créée n'appartient pas à la frontière de la configuration initiale, alors il se déduit d'un type initialement à la frontière en appliquant une règle « positive » du système d'inférence de la table 3.12 (p. 246), correspondant aux cas *(ii)* et *(iii)* du lemme 3.2.17 (p. 250).
Finalement, pour conclure, il nous restera à comparer $\mathcal{F}_\iota(s') \cup \mathcal{F}_\iota(r')$ avec $\mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(r)$, ce que nous ferons en un second temps. Nous verrons qu'une nouvelle frontière peut être créée par le remplacement d'un terme par un autre dans la configuration initiale, et que si le type de la frontière nouvellement créée n'appartient pas à la frontière de la configuration initiale, alors

il peut se déduire d'un type initialement à la frontière en appliquant une règle « négative » du système d'inférence, correspondant aux cas (iv) et (v) du lemme 3.2.17 (p. 250).
Commençons par étudier le passage au contexte de la réduction.

3.2.18 Lemme (Majoration de la frontière - cas [RED])
Soit $(\mathcal{C}_1, \mathcal{C}_2)$ une frontière.
Considérons une configuration $(s, E[r])$, où E est un contexte de réduction réduisant sous h et r un radical, qui se réduit en la configuration $(s', E[r'])$ et vérifie les deux conditions suivantes :

(i) la frontière de $(s, E[r])$ est incluse dans la frontière accessible à partir de la frontière $(\mathcal{C}_1, \mathcal{C}_2)$:

$$\begin{aligned} \mathcal{F}_{\mathsf{lo}}(s, E[r]) &\subseteq \mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}\,, \\ \mathcal{F}_{\mathsf{mo}}(s, E[r]) &\subseteq \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)}\,, \end{aligned}$$

(ii) le contexte de réduction E crée avec r' une nouvelle frontière :

$$\begin{aligned} \mathrm{F}^h_{\mathsf{lo}}(r) \cup \mathrm{F}^h_{\mathsf{mo}}(r) &= \emptyset\,, \\ \mathrm{F}^h_{\mathsf{lo}}(r') \cup \mathrm{F}^h_{\mathsf{mo}}(r') &\neq \emptyset\,. \end{aligned}$$

Alors la nouvelle frontière créée avec r' est incluse dans la frontière accessible à partir de la frontière $(\mathcal{C}_1, \mathcal{C}_2)$:

$$\begin{aligned} \mathrm{F}^h_{\mathsf{lo}}(r') &\subseteq \mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}\,, \\ \mathrm{F}^h_{\mathsf{mo}}(r') &\subseteq \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)}\,. \end{aligned}$$

Démonstration
On pose $(B, \tau) \stackrel{def}{=} r'.\mathrm{L}$. Comme E crée une nouvelle frontière avec r', on a $(B, \overline{\tau}) = r.\mathrm{L}$ et $h.\mathrm{I} = \overline{\tau}$.
On note $\mathcal{A}_\tau$ le majorant de $\mathcal{F}_\tau(s, E[r])$:

$$\mathcal{A}_\tau = \begin{cases} \mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)} & \text{si } \tau = \mathsf{lo}\,, \\ \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)} & \text{si } \tau = \mathsf{mo}\,. \end{cases}$$

Il s'agit de montrer que $\mathrm{F}^h_\tau(r') \subseteq \mathcal{A}_\tau$, soit $B \subseteq \mathcal{A}_\tau$, à partir de l'hypothèse $\mathcal{F}_\tau(s, E[r]) \subseteq \mathcal{A}_\tau$, soit $\mathcal{F}_\tau(s) \cup \mathcal{F}_\tau(E) \cup \mathcal{F}_\tau(r) \subseteq \mathcal{A}_\tau$.
Examinons les différents cas pour la réduction $(s, r) \to (s', r')$.
• $[\beta]$
La réduction est

$$(s, E[@^{(B,\overline{\tau})}(\lambda x^{(A\to B,\iota)}\, b, v)]) \to (s, E[b[v/x]])\,.$$

Distinguons deux cas, suivant que b est la variable x ou non.
◦ $b = x$
Dans ce cas, on a $r' = v$, si bien que B appartient à la frontière $\mathcal{F}_\tau(r)$, donc à $\mathcal{A}_\tau$.
◦ $b \neq x$
Dans ce cas, l'origine de r', soit τ, est égale à l'origine de b.
Si l'origine de l'abstraction ι est différente de τ, alors B appartient à la frontière $\mathcal{F}_\tau(r)$, donc à $\mathcal{A}_\tau$.
Si $\iota = \tau$, alors $A \to B$ appartient à la frontière $\mathcal{F}_\tau(r)$, donc à $\mathcal{A}_\tau$. De la propriété (ii) du lemme 3.2.17 (p. 250), on déduit que B appartient à $\mathcal{A}_\tau$.
• [REF]
La réduction serait

$$(s, E[\mathrm{ref}^{(B,\tau)}(v)]) \to ((s, l^{(B,\tau)} \mapsto v), E[l^{(B,\tau)}])\,,$$

où $l^{(B,\tau)} = \nu_{(B,\tau)}(s)$. Or on constate que E ne crée pas de nouvelle frontière avec r'.
• [REF-!]
La réduction est

$$(s, E[\mathrm{get}^{(B,\overline{\tau})}(l^{(\mathrm{Ref}(B),\iota)})]) \to (s, E[s(l^{(\mathrm{Ref}(B),\iota)})])\,.$$

Examinons les deux cas possibles pour l'origine ι de la référence.
◦ $\iota = \tau$
Le type $\mathrm{Ref}(B)$ appartient à $\mathcal{F}_\tau(r)$, donc à $\mathcal{A}_\tau$; de la propriété (iii) du lemme 3.2.17 (p. 250), on déduit que B appartient à $\mathcal{A}_\tau$.
◦ $\iota \neq \tau$
Le type B appartient à $\mathcal{F}_\tau(s)$, donc à $\mathcal{A}_\tau$.
• [REF-?]
La réduction serait

$$(s, E[\mathrm{set}^{(\mathrm{Unit},\tau)}(l^{(\mathrm{Ref}(B),\iota)}, v)]) \to ((s, l^{(\mathrm{Ref}(B),\iota)} \mapsto v), E[\mathrm{unit}^{(\mathrm{Unit},\tau)}])\,.$$

Or E ne crée pas de nouvelle frontière avec r'.
⌞

Venons-en maintenant aux différents axiomes de la relation de réduction. On traite à part la β-réduction, puisqu'elle nécessite une hypothèse supplémentaire, la cohérence des origines.

3.2.19 Lemme (Majoration de la frontière - cas $[\beta]$)
Soit $(\mathcal{C}_1, \mathcal{C}_2)$ une frontière.
Considérons une configuration (s, r), où r est un radical, qui se réduit en la

configuration (s', r'), et dont la frontière est incluse dans la frontière accessible à partir de la frontière $(\mathcal{C}_1, \mathcal{C}_2)$:

$$\begin{aligned} \mathcal{F}_{\mathsf{lo}}(s, r) &\subseteq \mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}\,, \\ \mathcal{F}_{\mathsf{mo}}(s, r) &\subseteq \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)}\,. \end{aligned}$$

Supposons que la réduction $(s, r) \to (s', r')$ forme un axiome $[\beta]$. Alors si la configuration (s, r) est d'origine cohérente, la frontière de (s', r') est incluse dans la frontière accessible à partir de la frontière $(\mathcal{C}_1, \mathcal{C}_2)$:

$$\begin{aligned} \mathcal{F}_{\mathsf{lo}}(s', r') &\subseteq \mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}\,, \\ \mathcal{F}_{\mathsf{mo}}(s', r') &\subseteq \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)}\,. \end{aligned}$$

Démonstration
La réduction est

$$(s, @^{(B,\iota_1)}(\lambda x^{(A\to B,\iota_2)}\, b, v)) \to (s, b[v/x])\,.$$

Il s'agit donc de montrer que la frontière de $b[v/x]$ est incluse dans la frontière accessible à partir de la frontière $(\mathcal{C}_1, \mathcal{C}_2)$.
Soit ι un élément quelconque de $\{\mathsf{mo}, \mathsf{lo}\}$.
On note $\mathcal{A}_\iota$ le majorant de $\mathcal{F}_\iota(s, r)$:

$$\mathcal{A}_\iota = \begin{cases} \mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)} & \text{si } \iota = \mathsf{lo}\,, \\ \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)} & \text{si } \iota = \mathsf{mo}\,. \end{cases}$$

Si $b = x$ ou $x \notin \mathbf{FV}(e)$, on a $\mathcal{F}_\iota(r') \subseteq \mathcal{F}_\iota(r)$, donc $\mathcal{F}_\iota(r') \subseteq \mathcal{A}_\iota$.
On peut donc supposer $b \neq x$ et $x \in \mathbf{FV}(e)$.
Comme (s, r) est d'origine cohérente, par la proposition 3.2.12 (p. 245), (s', r') l'est aussi ; du lemme 3.2.15 (p. 248), on obtient alors que $\mathcal{F}_\iota(r') = \mathcal{F}_\iota(b) \cup \mathcal{F}_\iota(v) \cup \mathrm{F}^m_\iota(v)$, où m est l'étiquette de b.
Comme $\mathcal{F}_\iota(b) \cup \mathcal{F}_\iota(v) \subseteq \mathcal{F}_\iota(r)$, le seul cas posant problème est lorsque $\mathrm{F}^m_\iota(v) \neq \emptyset$.
Supposons $\mathrm{F}^m_\iota(v) \neq \emptyset$, soit $\mathrm{F}^m_\iota(v) = \{A\}$, puisque A est le type de v. Soit τ l'origine de v. On a alors : $\iota = \tau$, $m.\mathrm{I} = \overline{\tau}$. Il s'agit de montrer que A appartient à $\mathcal{A}_\tau$. Examinons les deux cas possibles pour l'origine de l'application.

- $\iota_1 = \overline{\tau}$

A appartient à la frontière $\mathcal{F}_\tau(r)$, donc à $\mathcal{A}_\tau$.

- $\iota_1 = \tau$

Nécessairement l'origine ι_2 de l'abstraction est distincte de τ. Sinon, on aurait $x \in \mathbf{FV}(b)$, b d'origine $\overline{\tau}$, et $\iota_2 = \tau$, si bien que $\lambda x^{(A\to B,\iota_2)}\, b$ ne serait

pas d'origine cohérente, contradiction. Il s'ensuit que $A \to B$ appartient à $\mathcal{F}_{\overline{\tau}}(r)$, donc à $\mathcal{A}_{\overline{\tau}}$; de la propriété (iv) du lemme 3.2.17 (p. 250), on déduit que A appartient à $\mathcal{A}_\tau$.
⌞

Voyons les axiomes concernant les références.

3.2.20 Lemme (Majoration de la frontière - cas [REF], [REF- !] et [REF- ?])

Soit $(\mathcal{C}_1, \mathcal{C}_2)$ une frontière.
Considérons une configuration (s, r), où r est un radical, qui se réduit en la configuration (s', r'), et dont la frontière est incluse dans la frontière accessible à partir de la frontière $(\mathcal{C}_1, \mathcal{C}_2)$:

$$\begin{aligned} \mathcal{F}_{\mathsf{lo}}(s, r) &\subseteq \mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}\,, \\ \mathcal{F}_{\mathsf{mo}}(s, r) &\subseteq \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)}\,. \end{aligned}$$

Supposons que la réduction $(s, r) \to (s', r')$ forme un axiome [REF], [REF- !] ou [REF- ?]. *Alors la frontière de (s', r') est incluse dans la frontière accessible à partir de la frontière $(\mathcal{C}_1, \mathcal{C}_2)$:*

$$\begin{aligned} \mathcal{F}_{\mathsf{lo}}(s', r') &\subseteq \mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)}\,, \\ \mathcal{F}_{\mathsf{mo}}(s', r') &\subseteq \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)}\,. \end{aligned}$$

Démonstration
Soit ι un élément quelconque de $\{\mathsf{mo}, \mathsf{lo}\}$.
On note $\mathcal{A}_\iota$ le majorant de $\mathcal{F}_\iota(s, r)$:

$$\mathcal{A}_\iota = \begin{cases} \mathcal{A}_{(\mathcal{C}_1,\mathcal{C}_2)} & \text{si } \iota = \mathsf{lo}\,, \\ \mathcal{A}_{(\mathcal{C}_2,\mathcal{C}_1)} & \text{si } \iota = \mathsf{mo}\,. \end{cases}$$

Examinons les différents cas pour la réduction $(s, r) \to (s', r')$.
• [REF]
La réduction est

$$(s, \mathrm{ref}^m(v)) \to ((s, l^m \mapsto v), l^m)\,,$$

où $l^m = \nu_m(s)$.
Du lemme 3.2.16 (p. 249), on obtient l'inégalité suivante :

$$\mathcal{F}_\iota(s', r') \subseteq \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(v) \cup \mathrm{F}^m_\iota v\,,$$

soit

$$\mathcal{F}_\iota(s', r') \subseteq \mathcal{F}_\iota(s, r)\,.$$

• [REF- !]
La réduction est

$$(s, \mathrm{get}^{\,m}(l^n)) \to (s, s(l^n))\,.$$

On a :

$$\mathcal{F}_\iota(s', r') \subseteq \mathcal{F}_\iota(s)\,.$$

• [REF- ?]
La réduction est

$$(s, \mathrm{set}^{\,(\mathrm{Unit},\,\iota_1)}(l^{(\mathrm{Ref}(B),\,\iota_2)}, v)) \to ((s, l^{(\mathrm{Ref}(B),\,\iota_2)} \mapsto v), \mathrm{unit}^{(\mathrm{Unit},\,\iota_1)})\,.$$

Par le lemme 3.2.16 (p. 249), on a :

$$\mathcal{F}_\iota(s', r') \subseteq \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(v) \cup \mathrm{F}_\iota^{(\mathrm{Ref}(B),\,\iota_2)}(v)\,.$$

Comme $\mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(v) \subseteq \mathcal{A}_\iota$, il reste à montrer que $\mathrm{F}_\iota^{(\mathrm{Ref}(B),\,\iota_2)}(v) \subseteq \mathcal{A}_\iota$. Examinons donc le cas où $\mathrm{F}_\iota^{(\mathrm{Ref}(B),\,\iota_2)}(v) \neq \emptyset$.
Soit τ l'origine de v. On a alors $\iota = \tau$, $\mathrm{F}_\tau^{(\mathrm{Ref}(B),\,\iota_2)}(v) = \{B\}$, puisque v a pour type B, et $\tau \neq \iota_2$. Examinons les deux cas possibles pour l'origine de l'opérateur $\mathrm{set}(-,-)$, soit ι_1.
◦ $\iota_1 = \tau$
Le type $\mathrm{Ref}(B)$ appartient à $\mathcal{F}_{\overline{\tau}}(r)$, donc à $\mathcal{A}_{\overline{\tau}}$. De la propriété (v) du lemme 3.2.17 (p. 250), on déduit que B appartient à $\mathcal{A}_\tau$.
◦ $\iota_1 \neq \tau$
Le type B appartient à $\mathcal{F}_\tau(r)$, donc à $\mathcal{A}_\tau$.
⌞

Il est maintenant possible de démontrer le théorème 3.2.13 (p. 247) par récurrence.

Démonstration
Soit t une trace d'exécution telle que la configuration initiale t_0 soit d'origine cohérente.
En utilisant la proposition 3.2.12 (p. 245), on montre facilement par récurrence sur la position dans la trace que toute configuration t_i de t est d'origine cohérente.
On montre maintenant par récurrence sur la position dans la trace que toute configuration t_i de la trace t vérifie :

$$\begin{aligned} \mathcal{F}_{\mathsf{lo}}(t_i) &\subseteq \mathcal{A}_{(\mathcal{F}_{\mathsf{lo}}(t_0),\mathcal{F}_{\mathsf{mo}}(t_0))}\,, \\ \mathcal{F}_{\mathsf{mo}}(t_i) &\subseteq \mathcal{A}_{(\mathcal{F}_{\mathsf{mo}}(t_0),\mathcal{F}_{\mathsf{lo}}(t_0))}\,. \end{aligned}$$

• $i = 0$
Il suffit d'appliquer le lemme 3.2.17 (p. 250) (propriété (i)).
• $i > 0$
On suppose le résultat en $i-1$. La réduction $t_{i-1} \to t_i$ est égale à $(s, E[r]) \to (s', E[r'])$, où E est un contexte de réduction et r un radical.
Si $E = -$, il suffit d'appliquer les lemmes 3.2.19 (p. 252) et 3.2.20 (p. 254) pour conclure.
Supposons que E réduise sous h. Soit ι un élément quelconque de $\{\mathsf{mo}, \mathsf{lo}\}$. On note $\mathcal{A}_\iota$ le majorant de $\mathcal{F}_\iota(t_{i-1})$:

$$\mathcal{A}_\iota = \begin{cases} \mathcal{A}_{(\mathcal{F}_{\mathsf{lo}}(t_0), \mathcal{F}_{\mathsf{mo}}(t_0))} \text{si } \iota = \mathsf{lo}\,, \\ \mathcal{A}_{(\mathcal{F}_{\mathsf{mo}}(t_0), \mathcal{F}_{\mathsf{lo}}(t_0))} \text{si } \iota = \mathsf{mo}\,. \end{cases}$$

On a, par le lemme 3.2.18 (p. 251) :

$$\begin{aligned} \mathcal{F}_\iota(s, E[r]) &= \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(r) \cup \mathrm{F}^h_\iota(r)\,, \\ \mathcal{F}_\iota(s', E[r']) &= \mathcal{F}_\iota(s') \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(r') \cup \mathrm{F}^h_\iota(r')\,. \end{aligned}$$

Des lemmes 3.2.19 (p. 252) et 3.2.20 (p. 254), on déduit que

$$\mathcal{F}_\iota(s') \cup \mathcal{F}_\iota(r') \subseteq \mathcal{A}_\iota\,.$$

Il reste à montrer que $\mathrm{F}^h_\iota(r') \subseteq \mathcal{A}_\iota$.
Si $\mathrm{F}^h_\iota(r) \neq \emptyset$, nécessairement $\mathrm{F}^h_\iota(r') \subseteq \mathrm{F}^h_\iota(r)$, puisque r et r' ont même type, et c'est terminé. Sinon, si $\mathrm{F}^h_\iota(r') = \emptyset$, c'est terminé. Il reste le cas où le contexte de réduction E crée une nouvelle frontière avec r'. Du lemme 3.2.18 (p. 251), on déduit que $\mathrm{F}^h_\iota(r') \subseteq \mathcal{A}_\iota$.
On peut conclure.
⌞

3.2.4 Étude du confinement

Le théorème 3.2.13 (p. 247) est notre principal outil pour l'étude du confinement. Revenons à notre modèle d'étude, le code mobile s'exécutant dans un environnement local. Rappelons que nous modélisons le code mobile par un programme mobile, précisément une abstraction dont la variable représente l'environnement, et l'environnement local par un programme local. On note L le programme local, supposé de type L, et $\lambda x : L.\, \mathsf{M}[x]$ le programme mobile, où M est un contexte à une place, de type M (lorsqu'on attribue à la place le type L). Nous cherchons à déterminer une condition portant sur le type de l'environnement assurant le confinement des ressources dans

l'environnement lors de l'exécution du programme $(\lambda x : L.\, \mathsf{M}[x])\, \mathsf{L}$.
Pour réaliser cette étude, nous allons annoter le programme mobile par mo et l'environnement local par lo. Précisément, définissons le programme $\mathcal{M}(\mathsf{M}, \mathsf{L})$ par

$$\mathcal{M}(\mathsf{M}, \mathsf{L}) \stackrel{def}{=} @^{(M,\mathsf{mo})}(\langle \lambda x : L.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}})\,.$$

Le fait d'annoter les termes par leur origine permet de formaliser la notion de confinement, comme nous le voyons maintenant.

La réduction d'un radical détruit une valeur, excepté le cas particulier du radical créant et initialisant une référence (de la forme $\mathrm{ref}(v)$). Il est donc possible de décomposer tout radical de destruction sous la forme d'un destructeur, contexte formé d'un opérateur de destruction, appliqué à une place $-$, qu'on remplace par la valeur à détruire, et éventuellement à une place $*$, qu'on remplace par une valeur paramétrant la destruction. Le tableau suivant décrit précisément cette décomposition :

radical	destructeur	valeur détruite	valeur paramétrant
$@^n(\lambda x^m\, e, v)$	$@^n(-, *)$	$\lambda x^m\, e$	v
$\mathrm{get}^m(l^n)$	$\mathrm{get}^m(-)$	l^n	$\emptyset$
$\mathrm{set}^m(l^n, v)$	$\mathrm{set}^m(-, *)$	l^n	v

Nous sommes maintenant en mesure de formaliser la notion de confinement et d'accessibilité. On dit qu'un type apparaît dans un programme typé e si c'est le type d'une étiquette d'un sous-terme de $\langle e \rangle^\iota$, ι étant une information quelconque.

3.2.21 Définition (Confinement et accessibilité)
Soit L *un environnement et considérons un type* A.
Le type A *est* confiné *dans l'environnement* L *si*

(i) le type A *apparaît dans* L,

(ii) pour tout programme mobile M, *lors de l'exécution de* $\mathcal{M}(\mathsf{M}, \mathsf{L})$, *aucune valeur de type* A *et d'origine l'environnement n'est détruite par un destructeur d'origine le code mobile.*

Le type A *est* accessible *à partir de l'environnement* L *s'il existe un programme mobile* $\lambda x : L.\, \mathsf{M}[x]$ *tel que lors de l'exécution de* $\mathcal{M}(\mathsf{M}, \mathsf{L})$, *une valeur de type* A *et d'origine l'environnement est détruite par un destructeur d'origine le code mobile.*

On remarque qu'un type A apparaît nécessairement dans l'environnement L s'il est accessible. C'est qu'en effet aucune nouvelle étiquette n'est créée par

réduction. Ainsi, pour un type A apparaissant dans l'environnement, A est confiné si et seulement s'il est inaccessible.
Comme les destructions proscrites par le confinement se produisent à la frontière, on déduit du théorème 3.2.13 (p. 247) le corollaire suivant.

3.2.22 Corollaire (Caractérisation des types confinés et accessibles par les types sortants)

Soient L *un environnement de type* L *et* A *un type.*

(i) Si A *n'est pas un type sortant de* L *et si* A *apparaît dans* L*, alors* A *est confiné.*

(ii) Si A *est accessible à partir de* L*, alors* A *est un type sortant de* L*.*

Démonstration
Par définition, l'ensemble des types sortants de L est égal à $\mathcal{T}_+(L)$.
Soit t la trace d'exécution de $\mathcal{M}(\mathsf{M}, \mathsf{L})$. Comme pour la configuration initiale t_0, on a $\mathcal{F}_{\mathsf{lo}}(t_0) = \{L\}$ et $\mathcal{F}_{\mathsf{mo}}(t_0) = \emptyset$, on obtient par le théorème 3.2.13 (p. 247) que pour toute configuration t_i de t, $\mathcal{F}_{\mathsf{lo}}(t_i)$ est inclus dans $\mathcal{T}_+(L)$.
Si le type A est accessible à partir de L, alors il existe une configuration t_i telle que A appartient à $\mathcal{F}_{\mathsf{lo}}(t_i)$, si bien que A est un type sortant de L.
Si le type A apparaît dans l'environnement L, en contraposant l'implication précédente, on obtient la première implication.
∟

Nous montrons maintenant que la caractérisation précédente des types accessibles est optimale au sens où tout type sortant est aussi accessible à partir d'un certain environnement.

3.2.23 Théorème (Caractérisation des types sortants par les types accessibles)

Soit L *un type.*
Alors, pour tout type A *sortant de* L *différent de* Unit*, il existe un environnement* L *de type* L *tel que* A *est accessible à partir de* L*.*

On construit l'environnement et le programme mobile réalisant l'accès par récurrence structurelle sur le type de l'environnement. Prenons le cas où le type de l'environnement est un type fonctionnel $L_1 \to L_2$. Si le type A est un type sortant de $L_1 \to L_2$, autrement dit s'il apparaît dans $L_1 \to L_2$ en une occurrence positive ou sous la garde du constructeur Ref(−), alors deux cas sont envisageables lorsque A est distinct de $L_1 \to L_2$:
- ou bien A est un type entrant de L_1,
- ou bien A est un type sortant de L_2.

Dans le second cas, l'hypothèse de récurrence s'applique, dans le premier, non, si bien qu'on procède différemment : on montre qu'il est possible de décomposer L_1 jusqu'à obtenir un type L_1' admettant A comme type sortant. Formalisons dans un premier temps les propriétés liées à la décomposition du type de l'environnement, à l'aide du système d'inférence de la table 3.12 (p. 246).

3.2.24 Proposition
Soit $*$ un élément de $\{-,+\}$.
Considérons deux types distincts C et A. Alors :

(i) si C est égal à $C_1 \to C_2$, le jugement $C \vdash A : *$ est valide si et seulement si au moins l'un des jugements $C_1 \vdash A : \overline{*}$ et $C_2 \vdash A : *$ est valide.

(ii) si C est égal à $\mathrm{Ref}(D)$, le jugement $C \vdash A : *$ est valide si et seulement si au moins l'un des jugements $D \vdash A : \overline{*}$ et $D \vdash A : *$ est valide.

Démonstration
Commençons par une remarque utile par la suite : si le jugement $C \vdash A : *$ est valide, alors A est un sous-arbre de C, strict si $* = -$. La démonstration par récurrence structurelle sur la preuve de $C \vdash A : *$ est facile et omise.
Dans un premier temps, on montre par récurrence sur la preuve de $C \vdash A : *$ que si $C \neq A$, alors l'un des jugements $C' \vdash A : \ldots$ indiqués est valide, C' étant un sous-arbre immédiat de C.
• $[+]$
La preuve se réduit à l'axiome $C \vdash C : +$. La propriété est trivialement vérifiée.
• $[- \to]$
La preuve se conclut par la règle

$$\frac{C \vdash A \to B : \overline{*}}{C \vdash A : *}.$$

Distinguons deux cas suivant la nature de C.
◦ $C = C_1 \to C_2$
Si $C = A \to B$, nécessairement $* = -$, puis par décomposition, on a $C_1 = A$, et donc $C_1 \vdash A : \overline{*}$ est valide.
Si $C \neq A \to B$, alors de l'hypothèse de récurrence, on déduit qu'au moins l'un des jugements $C_1 \vdash A \to B : *$ et $C_2 \vdash A \to B : \overline{*}$ est valide, puis par application de la règle $[- \to]$, qu'au moins l'un des jugements $C_1 \vdash A : \overline{*}$ et $C_2 \vdash A : *$ est valide.
◦ $C = \mathrm{Ref}(D)$

De l'hypothèse de récurrence, on déduit qu'au moins l'un des jugements $D \vdash A \to B : *$ et $D \vdash A \to B : \overline{*}$ est valide, puis par application de la règle $[- \to]$, qu'au moins l'un des jugements $D \vdash A : \overline{*}$ et $D \vdash A : *$ est valide.

• Autres cas

Ce sont des variantes du cas précédent.

Venons-en maintenant à la réciproque. Nous la montrons sous la forme équivalente suivante : pour tous types C et A, si le jugement $C \vdash A : *$ est valide, alors les jugements suivants le sont aussi :

- pour tout type D, $D \to C \vdash A : *$ et $C \to D \vdash A : \overline{*}$,
- $\mathrm{Ref}(C) \vdash A : *$ et $\mathrm{Ref}(C) \vdash A : \overline{*}$.

On procède par récurrence structurelle sur la preuve de $C \vdash A : *$. À chaque fois, on considère un type D quelconque.

• $[+]$

La preuve se réduit à l'axiome $C \vdash C : +$. Prouvons donc les différents jugements :

$$(i) \qquad \dfrac{\dfrac{\emptyset}{C \to D \vdash C \to D : +}}{C \to D \vdash C : -} \quad ,$$

$$(ii) \qquad \dfrac{\dfrac{\emptyset}{D \to C \vdash D \to C : +}}{D \to C \vdash C : +} \quad ,$$

$$(iii) \qquad \dfrac{\dfrac{\emptyset}{\mathrm{Ref}(C) \vdash \mathrm{Ref}(C) : +}}{\mathrm{Ref}(C) \vdash C : *} \quad .$$

• $[- \to]$

La preuve se conclut par la règle

$$\dfrac{C \vdash A \to B : \overline{*}}{C \vdash A : *}.$$

De l'hypothèse de récurrence appliquée à $C \vdash A \to B : \overline{*}$, on déduit $C \to D \vdash A \to B : *$, $D \to C \vdash A \to B : \overline{*}$, $\mathrm{Ref}(C) \vdash A \to B : *$ et $\mathrm{Ref}(C) \vdash A \to B : \overline{*}$, puis $C \to D \vdash A : \overline{*}$, $D \to C \vdash A : *$, $\mathrm{Ref}(C) \vdash A : \overline{*}$

et $\mathrm{Ref}(C) \vdash A : *$.
• Autres cas
Ils sont analogues au précédent.
⌞

La démonstration du théorème 3.2.23 (p. 258) est principalement un exercice de programmation.
En premier lieu, nous montrons que chaque type contient au moins un programme convergent.

3.2.25 Lemme
Il existe une famille de programmes $(a_A)_A$, indexée par l'ensemble des types, ainsi qu'une famille de valeurs $(u_A)_A$ et une famille d'états-mémoire $(\sigma_A)_A$, telles que pour tout type A, $(\emptyset, a_A)$ converge vers (σ_A, u_A).

Démonstration
On procède par récurrence structurelle sur le type A.
• $A = \mathrm{Unit}$
On pose $a_A = u_A = \mathrm{unit}$ et $\sigma_A = \emptyset$.
• $A = A_1 \to A_2$
On pose $a_A = u_A = \lambda x : A_1 . a_{A_2}$ et $\sigma_A = \emptyset$.
• $A = \mathrm{Ref}(A_1)$
On pose $a_A = \mathrm{ref}(a_{A_1})$, $u_A = \nu_{A_1}(\sigma_{A_1})$ et $\sigma_A = (\sigma_{A_1}, u_A \mapsto u_{A_1})$.
⌞

Ce lemme nous est utile pour définir des valeurs par défaut.
Les notations suivantes facilitent la programmation.
Si e_1 et e_2 sont deux termes et x une variable qui n'est pas libre dans e_1, et de type A dans e_2, alors let $A\, x = e_1$ in e_2 déclare une variable locale de type A dans e_2 et l'initialise à e_1, et se définit par :

$$\text{let } A\, x = e_1 \text{ in } e_2 \stackrel{def}{=} (\lambda x : A . e_2)\, e_1 .$$

Si e_1 et e_2 sont deux termes, alors $e_1 ; e_2$ réalise la composition séquentielle de e_1 et e_2, et se définit par :

$$e_1 ; e_2 \stackrel{def}{=} (\lambda x\, e_2)\, e_1 ,$$

où x n'est pas une variable libre de e_2 (on a omis le type de x).
Si f et g ont pour type $A_1 \to A_2$ et $A_2 \to A_3$, alors $g \circ f$ réalise la composition fonctionnelle de g par f, et se définit par :

$$g \circ f \stackrel{def}{=} \lambda x : A_1 . g\, (f\, x) ,$$

où x n'est une variable libre ni de f ni de g.
Enfin, nous désignerons par $s_1 \cdot \ldots \cdot s_n$ l'état-mémoire s si les fonctions $(s_i)_i$ et l'état-mémoire s vérifient :

- pour tout i, s prolonge s_i,
- $(\operatorname{dom} s_i)_i$ forme une partition de s,
- pour tout i, l'ensemble $\{l \mid \exists\, j \leq i, A\,.\ l^A \in \operatorname{dom} s_j\}$ est un segment initial de l'ensemble des identificateurs.

Voyons maintenant précisément le cas des types entrants, avec ce lemme suivi d'une brève explication.

3.2.26 Lemme
Soient L et A deux types tels que A soit un type entrant de L.
Alors il existe deux types L' et L'', et deux programmes δ et γ, de type respectivement $L \to L'$ et $L' \to L$, vérifiant :

(i) L' *est un sous-arbre de* L,

(ii) L' *est égal à* $\mathrm{Ref}(L'')$ *ou à* $L'' \to B$ *pour un certain type* B,

(iii) L' *admet* A *comme type entrant,*

(iv) L'' *admet* A *comme type sortant,*

(v) les programmes δ *et* γ *sont des abstractions,*

(vi) pour toute configuration (s_1, v)*, où* s_1 *est un état-mémoire et* v *une valeur de type* L'*, il existe une valeur notée* u *et un état-mémoire noté* s *tels que*

(a) $(s_1, \gamma\, v)$ *s'évalue en* $(s_1 \cdot s, u)$,

(b) pour toute configuration $(s_2 \cdot s \cdot s_3, \delta\, u)$*, où* s_2 *et* s_3 *sont des états-mémoire, il existe un état-mémoire* s_4 *tel que* $(s_2 \cdot s \cdot s_3, \delta\, u)$ *s'évalue en* $(s_2 \cdot s \cdot s_3 \cdot s_4, v)$.

Aux effets de bord près, le lemme énonce que $\delta \circ \gamma$ est l'identité. En permettant de passer de L' à L, puis de revenir, ces deux programmes rendent possible l'utilisation de l'hypothèse de récurrence en L''. Avant de voir comment, démontrons ce lemme.

Démonstration
La démonstration se fait par récurrence sur L. Soit A un type. On montre que si $L \vdash A : -$, alors on peut définir deux types, L' et L'', et deux programmes, δ et γ vérifiant les conditions de l'énoncé.

- $L = \mathrm{Unit}$

Il n'existe pas de type A tel que $L \vdash A : -$.

- $L = L_1 \to L_2$

Supposons $L \vdash A : -$. Suivant la proposition 3.2.24 (p. 259), il existe deux possibilités, que nous détaillons.
◦ $L_1 \vdash A : +$
On définit L' par L, L'' par L_1, δ et γ par l'identité $\lambda x : L.\, x$.
◦ $L_2 \vdash A : -$
De l'hypothèse de récurrence appliquée à L_2, on obtient L', L'', δ et γ.
Soient $F = \lambda f : L.\, f\, a_{L_1}$ et $G = \lambda y : L_2.\, \lambda x : L_1.\, y$. On vérifie facilement que L', L'', $\delta \circ F$ et $G \circ \gamma$ conviennent.
• $L = \mathrm{Ref}(L_1)$
Supposons $L \vdash A : -$. Suivant la proposition 3.2.24 (p. 259), il existe deux possibilités, que nous détaillons.
◦ $L_1 \vdash A : +$
Comme précédemment, on définit L' par L, L'' par L_1, δ et γ par l'identité $\lambda x : L.\, x$.
◦ $L_1 \vdash A : -$
De l'hypothèse de récurrence appliquée à L_1, on obtient L', L'', δ et γ.
Soient $F = \lambda x : L.\, \mathrm{get}(x)$ et $G = \lambda x : L_1.\, \mathrm{ref}(x)$. On vérifie facilement que L', L'', $\delta \circ F$ et $G \circ \gamma$ conviennent.
⌞

Nous sommes désormais en mesure de démontrer le théorème 3.2.23 (p. 258). Dans cette démonstration, nous ne montrons pas que les traces d'exécution des programmes que nous définissons vérifient la propriété souhaitée. Il est facile de s'assurer informellement que c'est le cas : on raisonne en utilisant les conditions *(v)* et *(vi)* du lemme 3.2.26, que nous avons mises sous cette forme pour simplifier la vérification, ainsi que quelques propriétés intuitives des traces d'exécution. Manuellement, il serait très long de formaliser complètement le raisonnement.

Démonstration
On procède par récurrence sur L. Soit A un type différent de Unit. On montre que si $L \vdash A : +$, alors on peut définir un environnement local L de type L et un programme mobile $\lambda x : L.\, \mathsf{M}[x]$ tels que lors de l'exécution de $\mathcal{M}(\mathsf{M}, \mathsf{L})$, une valeur de type A et d'origine l'environnement est détruite par un destructeur d'origine le code mobile.
Le cas où A est égal à L étant trivial, on peut supposer A distinct de L.
• $L = \mathrm{Unit}$
On ne peut avoir $L \vdash A : +$.
• $L = L_1 \to L_2$
Supposons $L \vdash A : +$. Suivant la proposition 3.2.24 (p. 259), il existe deux possibilités, que nous détaillons.

◦ $L_2 \vdash A : +$
De l'hypothèse de récurrence appliquée à L_2, on obtient un environnement local L de type L_2 et un programme mobile $\lambda x : L_2.\, \mathsf{M}[x]$ tels que l'exécution de $\mathcal{M}(\mathsf{M}, \mathsf{L})$ entraîne la destruction attendue. On vérifie facilement que

$$\lambda x : L_1.\, \mathsf{L}$$

et

$$\lambda f : L.(\lambda x : L_2.\, \mathsf{M}[x])\,(f\, a_{L_1})$$

conviennent comme environnement local et programme mobile respectivement.
◦ $L_1 \vdash A : -$
Par le lemme précédent 3.2.26, on obtient deux types L' et L'', et deux programmes δ et γ, de type respectivement $L_1 \to L'$ et $L' \to L_1$, vérifiant les différentes conditions. Comme $L'' \vdash A : +$, en appliquant l'hypothèse de récurrence à L'', on dispose d'un environnement local L de type L'' et d'un programme mobile $\lambda x : L''.\, \mathsf{M}[x]$ tels que l'exécution de $\mathcal{M}(\mathsf{M}, \mathsf{L})$ entraîne la destruction souhaitée.
Envisageons deux cas, suivant le rapport entre L'' et L'.
◦ $L' = L'' \to B$
On définit le nouvel environnement par

$$\mathsf{L}' \stackrel{def}{=} \lambda x : L_1.(\delta\, x\, \mathsf{L}\,;\, a_{L_2})$$

et le nouveau programme mobile par

$$\lambda f : L.\, \mathsf{M}'[f] \stackrel{def}{=} \lambda f : L.\, f\,(\gamma\ \lambda x : L''.(\mathsf{M}[x]\,;\, a_B))\,.$$

Il est facile de vérifier que l'exécution de $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entraîne la destruction souhaitée.
◦ $L' = \mathrm{Ref}(L'')$
On définit le nouvel environnement par

$$\mathsf{L}' \stackrel{def}{=} \lambda x : L_1.(\mathrm{set}(\delta\, x, \mathsf{L})\,;\, a_{L_2})$$

et le nouveau programme mobile par

$$\begin{array}{rcl} \lambda f : L.\, \mathsf{M}'[f] & \stackrel{def}{=} & \lambda f : L.\, \mathrm{let}\ L'\, z = a_{L'}\ \mathrm{in} \\ & & \quad (f\,(\gamma\, z)\,;\, (\lambda x : L''.\, \mathsf{M}[x])\,\mathrm{get}(z))\,. \end{array}$$

Il est facile de vérifier que l'exécution de $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entraîne la destruction souhaitée.

• $L = \mathrm{Ref}(L_1)$
Supposons $L \vdash A : +$. Suivant la proposition 3.2.24 (p. 259), il existe deux possibilités, que nous détaillons.
◦ $L_1 \vdash A : +$
De l'hypothèse de récurrence appliquée à L_1, on obtient un environnement local L de type L_1 et un programme mobile $\lambda x : L_1.\, \mathsf{M}[x]$ tels que l'exécution de $\mathcal{M}(\mathsf{M}, \mathsf{L})$ entraîne la destruction attendue. On vérifie facilement que

$$\mathrm{ref}(\mathsf{L})$$

et

$$\lambda z : L.\, \mathrm{let}\ L_1\ x = \mathrm{get}(z)\ \mathrm{in}\ \mathsf{M}[x]$$

conviennent comme environnement local et programme mobile respectivement.
◦ $L_1 \vdash A : -$
Par le lemme précédent 3.2.26, on obtient deux types L' et L'', et deux programmes δ et γ, de type respectivement $L_1 \to L'$ et $L' \to L_1$, vérifiant les différentes conditions. Comme $L'' \vdash A : +$, en appliquant l'hypothèse de récurrence à L'', on dispose d'un environnement local L de type L'' et d'un programme mobile $\lambda x : L''.\, \mathsf{M}[x]$ tels que l'exécution de $\mathcal{M}(\mathsf{M}, \mathsf{L})$ entraîne la destruction attendue.
Envisageons deux cas, suivant le rapport entre L'' et L'.
◦ $L' = L'' \to B$
On définit le nouvel environnement par

$$\begin{aligned} \mathsf{L}' \quad \stackrel{def}{=} \quad & \mathrm{let}\ L\ z = a_L\ \mathrm{in} \\ & (\mathrm{set}(z, \gamma\ \lambda x : L''.\, \delta\, \mathrm{get}(z)\, \mathsf{L})\,;\, z) \end{aligned}$$

et le nouveau programme mobile par

$$\begin{aligned} \lambda z : L.\, \mathsf{M}'[z] \quad \stackrel{def}{=} \quad & \lambda z : L.\, \mathrm{let}\ L_1\ y = \mathrm{get}(z)\ \mathrm{in} \\ & (\mathrm{set}(z, \gamma\ \lambda x : L''.(\mathsf{M}[x]\,;\, a_B))\,;\, \delta\, y\, a_{L''})\,. \end{aligned}$$

Il est facile de vérifier que l'exécution de $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entraîne la destruction souhaitée.
◦ $L' = \mathrm{Ref}(L'')$
On définit le nouvel environnement par

$$\mathsf{L}' \quad \stackrel{def}{=} \quad \mathrm{let}\ L'\ y = \mathrm{ref}(\mathsf{L})\ \mathrm{in}\ \mathrm{ref}(\gamma\, y)$$

et le nouveau programme mobile par

$$\lambda z : L.\, \mathsf{M}'[z] \quad \stackrel{def}{=} \quad \lambda z : L.\, \text{let } L''\, y = \text{get}(\delta\, \text{get}(z)) \text{ in } \mathsf{M}[y]\,.$$

Il est facile de vérifier que l'exécution de $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entraîne la destruction souhaitée.

⌞

Ici s'achève l'étude générale du confinement. En utilisant un langage annoté particulier, dans lequel chaque terme porte, en plus de son type, son origine, le code mobile ou le code local, nous avons pu définir d'une part, une notion de frontière, locale et dynamique, lieu des interactions entre le code mobile et le code local, d'autre part, la propriété de confinement de manière rigoureuse. L'étude de la frontière a permis de dégager un critère, relatif au type de l'environnement local et permettant d'affirmer si un type donné est confiné dans l'environnement local. Il se résume ainsi :

> si le type n'apparaît dans le type de l'environnement local ni en une occurrence positive ni sous la garde du constructeur Ref(−), alors il est confiné dans l'environnement local.

Bien sûr, si le type apparaît en une occurrence positive ou sous la garde du constructeur Ref(−), il est impossible d'affirmer quoi que ce soit. Cependant, on a montré que dans ce cas il existe un environnement local à partir duquel le type considéré est accessible par le code mobile. Autrement dit, nous avons obtenu le meilleur critère possible à partir du type de l'environnement local. De plus, il est évidemment décidable : étant donné un type L pour l'environnement local et un type à tester A, la question de savoir si A apparaît dans L en une occurrence positive ou sous la garde du constructeur Ref(−) est décidable. C'est à rapporter au problème général suivant, qui lui est indécidable :

- données :
  un environnement local L typé et un type A,
- question :
  le type A est-il accessible à partir de l'environnement local L ?

En effet, si ce problème était décidable, le problème de la terminaison serait décidable[12]. En effet, considérons un programme e et appliquons la procédure de décision supposée au type A et à l'environnement $e\,;\, a_A$, où a_A est

[12] On présume évidemment que le problème de la terminaison est indécidable pour notre langage.

le programme convergent de type A donné par le lemme 3.2.25 (p. 261). On constate que e termine si et seulement si le code mobile accède à une valeur de type A.

Revenons maintenant aux questions de sécurité, précisément au contrôle des accès. Nous avons vu en introduction que le confinement des ressources doit s'accompagner du contrôle de l'usage des ressources au sein du code local. C'est cette question que nous abordons maintenant.

3.2.5 De l'instrumentation au confinement

Nous distinguons le code instrumenté de celui qui ne l'est pas, de même le code à protéger de celui qui ne l'est pas. À cette fin, nous introduisons deux étiquettes, $\bot$ et $\top$, auxquelles nous attribuons la signification suivante, selon qu'elles annotent un constructeur ou un destructeur du langage :

	constructeur	destructeur
$\bot$	pas de protection	pas d'instrumentation
$\top$	protection	instrumentation

Dans un premier temps, nous développons un système de types de manière à assurer que l'exécution de tout programme y admettant un type vérifie la propriété suivante :

> un destructeur étiqueté par $\bot$ n'est jamais appliqué à une valeur étiquetée par $\top$,

ou encore en ordonnant la paire $\{\bot, \top\}$ par $\bot < \top$,

> un destructeur étiqueté par i détruit une valeur étiquetée par j seulement si $j \leq i$.

Ce système garantit ainsi la correction de l'instrumentation. Dans un second temps, nous utilisons la relation de sous-typage suivant le principe « convertir pour confiner » pour résoudre la question du confinement.

Nous développons un système utilisant des types étiquetés, dits *sécuritaires*. Leur ensemble est engendré par la grammaire suivante :

$$\begin{array}{rcll} A & ::= & \mathrm{Unit} & \text{(type singleton)} \\ & | & A \xrightarrow{\bot} A \mid A \xrightarrow{\top} A & \text{(types fonctionnels)} \\ & | & \mathrm{Ref}^{\bot}(A) \mid \mathrm{Ref}^{\top}(A) & \text{(types des références)}\,. \end{array}$$

Une valeur étiquetée par $\bot$ admettra un type étiqueté par $\bot$, et de même pour $\top$. Détaillons le langage annoté que nous utilisons, et évidemment conforme à la définition générale donnée dans le paragraphe 3.2.2 (p. 233).

Rappelons que les étiquettes annotant les opérateurs d'un tel langage sont formées d'un type (sans annotation) et d'une information. Se limiter à l'information indiquant l'instrumentation ou la protection est insuffisant dans notre cas si on souhaite déterminer le type sécuritaire à partir de la seule étiquette d'un terme. Par exemple, au programme $\lambda x^{(A,\top)}\, x$, où A est un type non étiqueté, on pourrait attribuer n'importe quel type $A' \xrightarrow{\top} A'$, où A' est une annotation du type A. Aussi, on prendra pour information un couple, formé d'un type sécuritaire et d'une valeur, $\bot$ ou $\top$. Comme le type sécuritaire permet de retrouver le type non étiqueté, simplement en effaçant les étiquettes, il suffit pratiquement de prendre pour étiquettes les couples formés d'un type sécuritaire et d'une valeur $\bot$ ou $\top$; si m est une telle étiquette, on notera $m.\mathrm{T}$ le type sécuritaire et $m.\mathrm{I}$ la valeur $\bot$ ou $\top$ vérifiant $m = (m.\mathrm{T}, m.\mathrm{I})$. Nous verrons par la suite que cette définition des étiquettes permet de déterminer le type minimal d'un terme. Désormais, on appelle un terme étiqueté de ce langage annoté un terme *instrumenté*.

Le système de types est décrit par le système d'inférence de la table 3.13 (p. 269) ; il permet de définir inductivement l'ensemble des jugements de typage valides, un jugement de typage étant de la forme $\Gamma \vdash e : A$, où Γ est un environnement, e un terme instrumenté et A un type sécuritaire, ces derniers respectivement sujet et objet du jugement « a pour type sécuritaire » dans l'environnement Γ. On remarque que si on n'utilise qu'une seule valeur $\bot$ ou $\top$ pour les étiquettes des types sécuritaires et des termes, on retrouve les règles de typage du système 3.9 (p. 235). La seule différence concerne les règles applicables aux destructeurs, où la contrainte concernant les étiquettes vise le respect du principe énoncé plus haut : l'information associée à un destructeur, qui indique s'il est instrumenté ou non, est comparée à l'étiquette du type du terme à détruire ; comme l'information associée à un constructeur coïncide avec l'étiquette de son type, on comprend que le principe est bien appliqué.
Nous aurons besoin par la suite d'annoter entièrement par $\bot$ un programme non étiqueté et typé. Précisément, notons $\langle A \rangle^{\alpha}$ le type sécuritaire obtenu en étiquetant par α tous les constructeurs de types, $- \to -$ ou $\mathrm{Ref}(-)$, du type non étiqueté A. Associons à tout programme e non étiqueté et typé le programme $\langle\!\langle e \rangle\!\rangle^{\bot}$, e coloré en $\bot$, dont l'étiquetage n'utilise que $\bot$, aussi bien pour les types sécuritaires que pour les étiquettes des constructeurs et des destructeurs, de la manière suivante :

$$\langle\!\langle e \rangle\!\rangle^{\bot} \stackrel{def}{=} G(\langle e \rangle^{\bot}) \,,$$

$$\frac{\emptyset}{\Gamma \vdash x \,:\, \Gamma(x)}\,[\text{intro-Var}] \quad (x \in \operatorname{dom}\Gamma)$$

$$\frac{\Gamma.(x \,:\, A) \vdash e \,:\, B}{\Gamma \vdash \lambda x^m\, e \,:\, A \xrightarrow{\alpha} B}\,[\text{intro-}\to] \quad \left(\begin{array}{l} A \xrightarrow{\alpha} B = m.\mathrm{T} \\ \alpha = m.\mathrm{I} \end{array}\right)$$

$$\frac{\Gamma \vdash f \,:\, A \xrightarrow{\alpha} B \quad \Gamma \vdash a \,:\, A}{\Gamma \vdash @^m(f,a) \,:\, B}\,[\text{élim-}\to] \quad \left(\begin{array}{l} B = m.\mathrm{T} \\ \alpha \leq m.\mathrm{I} \end{array}\right)$$

$$\frac{\emptyset}{\Gamma \vdash \mathrm{unit}^m \,:\, \mathrm{Unit}}\,[\text{intro-Unit}] \quad (\mathrm{Unit} = m.\mathrm{T})$$

$$\frac{\emptyset}{\Gamma \vdash l^m \,:\, \mathrm{Ref}^\alpha(A)}\,[\text{intro-Ref/loc}] \quad \left(\begin{array}{l} \mathrm{Ref}^\alpha(A) = m.\mathrm{T} \\ \alpha = m.\mathrm{I} \end{array}\right)$$

$$\frac{\Gamma \vdash e \,:\, A}{\Gamma \vdash \mathrm{ref}^m(e) \,:\, \mathrm{Ref}^\alpha(A)}\,[\text{intro-Ref/new}] \quad \left(\begin{array}{l} \mathrm{Ref}^\alpha(A) = m.\mathrm{T} \\ \alpha = m.\mathrm{I} \end{array}\right)$$

$$\frac{\Gamma \vdash e \,:\, \mathrm{Ref}^\alpha(A)}{\Gamma \vdash \mathrm{get}^m(e) \,:\, A}\,[\text{élim-Ref/ !}] \quad \left(\begin{array}{l} A = m.\mathrm{T} \\ \alpha \leq m.\mathrm{I} \end{array}\right)$$

$$\frac{\Gamma \vdash e_1 \,:\, \mathrm{Ref}^\alpha(A) \quad \Gamma \vdash e_2 \,:\, A}{\Gamma \vdash \mathrm{set}^m(e_1,e_2) \,:\, \mathrm{Unit}}\,[\text{élim-Ref/ ?}] \quad \left(\begin{array}{l} \mathrm{Unit} = m.\mathrm{T} \\ \alpha \leq m.\mathrm{I} \end{array}\right)$$

Tab. 3.13 – Système de types sécuritaires du langage instrumenté

$$\frac{\emptyset}{(\mathrm{Unit}, \mathrm{Unit})}$$

$$\frac{(A_2, A_1) \quad (B_1, B_2)}{(A_1 \xrightarrow{\alpha_1} B_1, A_2 \xrightarrow{\alpha_2} B_2)} \quad (\alpha_1 \leq \alpha_2)$$

$$\frac{\emptyset}{(\mathrm{Ref}^{\alpha_1}(A), \mathrm{Ref}^{\alpha_2}(A))} \quad (\alpha_1 \leq \alpha_2)$$

Tab. 3.14 – Types sécuritaires : relation de sous-typage

où G est défini par l'équation récursive suivante :

$$G(f^{(A,\alpha)}(e_j)_j) = f^{(\langle A \rangle^\perp, \alpha)}(G(e_j))_j \,.$$

Il est facile de vérifier que si e a pour type A, alors $\langle\!\langle e \rangle\!\rangle^\perp$ a pour type $\langle A \rangle^\perp$.

Considérons maintenant un type sécuritaire A dont l'étiquette est $\top$ et que nous notons $B^\top$. On note $B^\perp$ le type obtenu en modifiant l'étiquette de A en $\perp$. Soit $C_{[\,]}$ un programme instrumenté à un paramètre, admettant un type sécuritaire dans le système précédent lorsqu'on attribue le type $B^\top$ au paramètre. En utilisant la sémantique opérationnelle précédemment définie pour le langage annoté (cf. tab. 3.10 (p. 237)), on verra par la suite que pour toute valeur v de type $B^\top$, l'exécution de $C[v]$ n'entraîne pas de destruction interdite, c'est-à-dire impliquant un destructeur non instrumenté et une valeur à protéger. Il est clair que pour toute valeur v de type $B^\perp$, l'exécution de $C[v]$ n'entraîne pas non plus de destruction interdite. Autrement dit, on peut considérer que $B^\perp$ est un sous-type de $B^\top$.

Précisément, la relation de sous-typage entre les types sécuritaires est engendrée inductivement par le système d'inférence de la table 3.14 (p. 270). Les jugements sont des couples (A, B), où A et B sont des types sécuritaires : si le jugement (A, B) est valide, on dit que A est un sous-type de B, et on le note $A \leq B$. On remarque les propriétés classiques : pour le constructeur $- \to -$, contravariance pour l'ensemble de départ, covariance pour l'ensemble d'arrivée, pour le constructeur $\mathrm{Ref}(-)$, invariance. On vérifie facilement que la relation de sous-typage est une relation d'ordre partielle. Pour l'utiliser, on ajoute au système de types la règle de conversion suivante :

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e : B}\,[\leq] \quad (A \leq B)\,.$$

Le système de types vérifie les propriétés habituelles. Noter que l'unicité du type est remplacée par l'existence d'un type minimal.

3.2.27 Proposition (Propriétés du système de types sécuritaires)
Soit Γ *un environnement de typage. Considérons un terme instrumenté* a *de type sécuritaire* A *dans l'environnement* Γ *:*

$$\Gamma \vdash a : A\,.$$

Lemme d'affaiblissement
Considérons un environnement Γ' *prolongeant* Γ*. Alors le jugement de typage* $\Gamma' \vdash a : A$ *est valide.*

Lemme de la base
Considérons l'environnement $\Gamma_{|\mathbf{FV}(a)}$ *obtenu en restreignant* Γ *aux variables libres de* a*. Alors le jugement de typage* $\Gamma_{|\mathbf{FV}(a)} \vdash a : A$ *est valide.*

Existence d'un plus petit type
Si a *est une variable* x*, alors* a *admet un plus petit type (pour la relation de sous-typage) dans l'environnement* Γ*, soit* $\Gamma(x)$*.*
Sinon, soit M *le type donné par l'étiquette de* a *:*

$$M = a.\mathrm{L}.\mathrm{T}\,.$$

Alors M *est le plus petit type de* a *dans l'environnement* Γ*.*

Lemme des substitutions
Soient x *une variable du domaine de* Γ*, et* b *un terme instrumenté de type* $\Gamma(x)$*. Alors le terme* $a[b/x]$ *a pour type* A *dans l'environnement* Γ *:*

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : \Gamma(x)}{\Gamma \vdash a[b/x] : A} \quad (x \in \operatorname{dom}\Gamma)\,.$$

Lemme de la décomposition
Supposons que a *soit un programme se décomposant en un radical* r *dans un contexte de réduction* E *:*

$$a = E[r]\,.$$

Alors r *est typé, et si on attribue à la place* $-$ *le plus petit type de* r*,* E *admet pour type* A*.*

Démonstration
• Affaiblissement, base
La démonstration est analogue à celle de la proposition 3.2.1 (p. 226).
• Plus petit type
On montre très facilement par récurrence structurelle sur la preuve de $\Gamma \vdash a : A$ que a admet comme plus petit type dans l'environnement Γ celui qui est indiqué par la proposition.
• Substitutions
La démonstration, classique, se fait par récurrence structurelle sur a, et ne présente aucune difficulté.
• Décomposition
On procède par récurrence structurelle sur E. Seul cas original, le contexte de réduction vide.
Supposons $E = -$. Comme r est égal à a, r est typé ; quant au contexte de réduction E, il admet pour type le plus petit type de r, qui est le plus petit type de a, et est donc un sous-type de A.
⌞

Définissons maintenant la sémantique opérationnelle de notre langage. Pour simplifier, nous ne modifions pas la sémantique opérationnelle définie par le système de la table 3.10 (p. 237). Ce choix correspond à une interprétation en termes d'accessibilité. Pour un destructeur, l'étiquette $\top$ signifie la possibilité d'accéder aux valeurs devant être protégées, étiquetées par $\top$. Une approche réaliste consisterait à considérer qu'un destructeur réalise tout d'abord un test, puis s'il est positif, réalise la destruction habituelle, sinon arrête l'exécution ou lance une exception.
Énonçons la principale propriété permettant de relier la sémantique statique (le système de types) à la sémantique opérationnelle. On dit qu'une configuration (s, e) est typée pour la sécurité si elle est typée dans le système de types sécuritaires, autrement dit si

- pour toute référence l^m du domaine de s, la valeur $s(l^m)$ admet pour type sécuritaire le type A vérifiant
$$m.\mathrm{T} = \mathrm{Ref}^{m.\mathrm{I}}(A)\,,$$
- le terme de contrôle e admet un type sécuritaire.

3.2.28 Théorème (Réduction du sujet : préservation du typage)
Soit (s_1, e_1) une configuration typée pour la sécurité. Alors si (s_1, e_1) se réduit en (s_2, e_2), la configuration (s_2, e_2) est typée pour la sécurité. De plus, le plus petit type sécuritaire de e_2 est un sous-type du plus petit type sécuritaire de e_1.

Démonstration
On raisonne par récurrence structurelle sur la preuve de la réduction $(s_1, e_1) \to (s_2, e_2)$. On suppose que la configuration (s_1, e_1) est typée pour la sécurité, et on note M le plus petit type de e_1.
• $[\beta]$
La réduction est

$$(s_1, @^m(\lambda x^n\, a, v)) \to (s_1, a[v/x])\,.$$

La preuve de $\emptyset \vdash e_1 \,:\, M$ se termine ainsi :

$$\dfrac{\dfrac{\dfrac{\emptyset.(x \,:\, A') \vdash a \,:\, M'}{\emptyset \vdash \lambda x^n\, a \,:\, A' \xrightarrow{\alpha'} M'}}{\vdots} \qquad \emptyset \vdash v \,:\, A}{\emptyset \vdash @^m(\lambda x^n\, a, v) \,:\, M}$$

où $A \leq A'$, $M' \leq M$ et $\alpha' \leq \alpha$.
De ces inégalités et du lemme des substitutions, on déduit que $a[v/x]$ a pour type M, si bien que son plus petit type est un sous-type de M.
De plus, $(s_1, a[v/x])$ est typée pour la sécurité.
• [REF]
La réduction est

$$(s_1, \mathrm{ref}^m(v)) \to ((s_1, l^m \mapsto v), l^m)\,,$$

où l^m est égal à $\nu_m(s)$. Du jugement $\emptyset \vdash \mathrm{ref}^m(v) \,:\, M$, on déduit que v admet pour type A défini par $m.\mathrm{T} = \mathrm{Ref}^{m.\mathrm{I}}(A)$. Il en résulte que $((s_1, l^m \mapsto v), l^m)$ est typée pour la sécurité.
Enfin, $\mathrm{ref}^m(v)$ et l^m ont le même plus petit type, $m.\mathrm{T}$.
• [REF-!]
La réduction est

$$(s_1, \mathrm{get}^m(l^n)) \to (s_1, s_1(l^n))\,.$$

Évidemment, la configuration $(s_1, s_1(l^n))$ est typée pour la sécurité. Montrons que $s_1(l^n)$ admet M pour type.
Par hypothèse, on sait que $s_1(l^n)$ admet pour type A vérifiant $n.\mathrm{T} = \mathrm{Ref}^{n.\mathrm{I}}(A)$.

La preuve de $\emptyset \vdash \mathrm{get}^m(l^n) : M$ est ainsi formée

$$\dfrac{\dfrac{\emptyset}{\emptyset \vdash l^n : \mathrm{Ref}^{n.\mathrm{I}}(M)} \\ \vdots \\ \emptyset \vdash l^n : \mathrm{Ref}^{\alpha}(M)}{\emptyset \vdash \mathrm{get}^m(l^n) : M}$$

avec $n.\mathrm{T} = \mathrm{Ref}^{n.\mathrm{I}}(M)$. Il s'ensuit que $M = A$.

• [REF-?]
La réduction est

$$(s_1, \mathrm{set}^m(l^n, v)) \rightarrow ((s_1, l^n \mapsto v), \mathrm{unit}^m)\,.$$

On doit montrer que v admet pour type A défini par $n.\mathrm{T} = \mathrm{Ref}^{n.\mathrm{I}}(A)$. Or il existe un type B tel que la preuve de $\emptyset \vdash \mathrm{set}^m(l^n, v) : \mathrm{Unit}$ est formée ainsi :

$$\dfrac{\dfrac{\dfrac{\emptyset}{\emptyset \vdash l^n : \mathrm{Ref}^{n.\mathrm{I}}(B)} \\ \vdots \\ \emptyset \vdash l^n : \mathrm{Ref}^{\alpha}(B)} \qquad \emptyset \vdash v : B}{\emptyset \vdash \mathrm{set}^m(l^n, v) : \mathrm{Unit}}$$

avec $n.\mathrm{T} = \mathrm{Ref}^{n.\mathrm{I}}(B)$, ce qui implique $B = A$ et donc que v a pour type A.

• [RED]
La réduction est $(s_1, E[r_1]) \rightarrow (s_2, E[r_2])$, déduite de $(s_1, r_1) \rightarrow (s_2, r_2)$, et où E est un contexte de réduction et r_1 un radical.
Par le lemme de décomposition, on obtient que r_1 est typé et que E a pour type M si on attribue à la place $-$ le plus petit type de r_1. En appliquant l'hypothèse de récurrence à la réduction $(s_1, r_1) \rightarrow (s_2, r_2)$, comme (s_1, r_1) est typée pour la sécurité, on obtient que s_2 vérifie la condition de typage et que le plus petit type de r_2 est un sous-type du plus petit type de r_1. Par le lemme des substitutions, on obtient finalement que $E[r_2]$ admet pour type M, et on peut conclure.

⌞

Il résulte de ce théorème fondamental que le système de types sécuritaires est sûr.

3.2.29 Corollaire (Sûreté du typage)
Considérons un programme instrumenté admettant un type sécuritaire. Alors, lors de son exécution, aucun destructeur étiqueté par $\bot$ ne détruit une valeur étiquetée par $\top$.

Démonstration
Par le théorème de réduction du sujet, toute configuration de la trace d'exécution est typée pour la sécurité. Si lors de l'exécution, un destructeur étiqueté par $\bot$ détruisait une valeur étiquetée par $\top$, alors le radical en question n'admettrait pas de type sécuritaire, ce qui constituerait une contradiction, par le lemme de la décomposition.
⌞

Autrement dit, la vérification statique de l'instrumentation entraîne la vérification dynamique de la politique de sécurité.

Comment appliquer ce système de types dans le cas du code mobile ? Commençons par modéliser l'exécution du code mobile appelant l'environnement local avec le langage instrumenté. Comme le code mobile ne peut pas être instrumenté et ne contient pas de ressources à protéger, il ne contient que l'étiquette $\bot$. Précisément, si $\lambda x : L.\, \mathsf{M}[x]$ est le programme mobile de type $L \to M$, nous le représentons par le programme

$$\langle\!\langle \lambda x : L.\, \mathsf{M}[x] \rangle\!\rangle^{\bot},$$

de type

$$\langle L \rangle^{\bot} \xrightarrow{\bot} \langle M \rangle^{\bot}.$$

Quant au code local, il peut être instrumenté et contenir des ressources à protéger, si bien qu'il utilise les deux étiquettes $\bot$ et $\top$ suivant le principe déjà énoncé :

	constructeur	destructeur
$\bot$	pas de protection	pas d'instrumentation
$\top$	protection	instrumentation

On note $\mathsf{L_S}$ l'environnement local ainsi étiqueté. On suppose évidemment qu'il admet un type sécuritaire, qu'on note L_S. On sait donc que l'exécution de l'environnement local seul est sûre. Qu'en est-il de l'exécution du programme mobile appelant l'environnement local, autrement dit du programme

$$@^{(\langle M \rangle^{\bot}, \bot)}(\langle\!\langle \lambda x : L.\, \mathsf{M}[x] \rangle\!\rangle^{\bot}, \mathsf{L_S})$$

qui nous sert à modéliser cette exécution ? Une condition suffisante est évidemment que ce programme admette un type sécuritaire. Compte tenu des

règles de typage, il vient que ce programme admet un type sécuritaire si et seulement le type de l'environnement L_S est un sous-type du type $\langle L \rangle^{\perp}$, le même type que L_S si ce n'est que les étiquettes $\top$ ont été remplacées par des étiquettes $\perp$. La proposition suivante définit une condition équivalente, à l'aide des types sortants. Étant donné un type sécuritaire C, on note $\downarrow (C)$ le type non étiqueté obtenu en effaçant les étiquettes.

3.2.30 Proposition (Conversion et types sortants)
Soit C un type sécuritaire.
Alors C est un sous-type de $\langle \downarrow (C) \rangle^{\perp}$ si et seulement si tout type sortant de C et différent de Unit *a pour étiquette $\perp$.*

On définit les types sortants et entrants de C comme pour les types non étiquetés. Un type A est un type sortant de C s'il apparaît dans C en une occurrence positive ou sous la garde d'un constructeur $\mathrm{Ref}^{\alpha}(-)$; un type A est un type entrant de C s'il apparaît dans C en une occurrence négative ou sous la garde d'un constructeur $\mathrm{Ref}^{\alpha}(-)$. Formellement, on utilise le système d'inférence de la table 3.12 (p. 246), où chaque constructeur de types, $- \to -$ ou $\mathrm{Ref}(-)$, doit cependant être remplacé par un constructeur étiqueté. On rappelle qu'un type A est un type sortant de C si le jugement $C \vdash A : +$ est valide, alors que c'est un type entrant de C si le jugement $C \vdash A : -$ est valide.
On utilise dans la démonstration la proposition 3.2.24 (p. 259), qui permet de relier un jugement $C \vdash A : \ldots$ à un jugement $C' \vdash A : \ldots$, où C' est un sous-arbre de C, lorsque C est distinct de A.

Démonstration
Nous allons montrer non seulement la propriété de l'énoncé, mais aussi une propriété duale. Précisément, nous montrons les deux équivalences suivantes :

$$C \leq \langle \downarrow (C) \rangle^{\perp} \quad \Leftrightarrow \quad \forall\, A \neq \mathrm{Unit}\,.\; C \vdash A : + \Rightarrow A.\mathrm{I} = \perp\,,$$
$$\langle \downarrow (C) \rangle^{\perp} \leq C \quad \Leftrightarrow \quad \forall\, A \neq \mathrm{Unit}\,.\; C \vdash A : - \Rightarrow A.\mathrm{I} = \perp\,.$$

Commençons par montrer simultanément les deux implications de gauche à droite par récurrence sur C. On considère un type A distinct de Unit.
• $C = \mathrm{Unit}$
C'est trivialement vérifié.
• $C = C_1 \xrightarrow{\alpha} C_2$
Voyons la première implication.
Supposons $C \leq \langle \downarrow (C) \rangle^{\perp}$, ce qui implique par définition du sous-typage,

$\alpha = \bot$, $C_2 \leq \langle\downarrow (C_2)\rangle^\bot$ et $\langle\downarrow (C_1)\rangle^\bot \leq C_1$.
Supposons aussi $C \vdash A : +$.
Si $A = C$, alors A a pour étiquette α, soit $\bot$.
Sinon, on peut appliquer la proposition 3.2.24 (p. 259).
Si $C_1 \vdash A : -$, en appliquant l'hypothèse de récurrence à C_1, on obtient $A.\mathrm{I} = \bot$.
Si $C_2 \vdash A : +$, en appliquant l'hypothèse de récurrence à C_2, on obtient $A.\mathrm{I} = \bot$.
Voyons la seconde implication.
Supposons $\langle\downarrow (C)\rangle^\bot \leq C$, ce qui implique par définition du sous-typage, $C_1 \leq \langle\downarrow (C_1)\rangle^\bot$ et $\langle\downarrow (C_2)\rangle^\bot \leq C_2$.
Supposons aussi $C \vdash A : -$.
Nécessairement, $A \neq C$, et on peut appliquer la proposition 3.2.24 (p. 259).
Si $C_1 \vdash A : +$, en appliquant l'hypothèse de récurrence à C_1, on obtient $A.\mathrm{I} = \bot$.
Si $C_2 \vdash A : -$, en appliquant l'hypothèse de récurrence à C_2, on obtient $A.\mathrm{I} = \bot$.

- $C = \mathrm{Ref}^\alpha(D)$

Voyons la première implication.
Supposons $C \leq \langle\downarrow (C)\rangle^\bot$, ce qui implique par définition du sous-typage, $\alpha = \bot$, $D = \langle\downarrow (D)\rangle^\bot$.
Supposons aussi $C \vdash A : +$.
Si $A = C$, alors A a pour étiquette α, soit $\bot$.
Sinon, on peut appliquer la proposition 3.2.24 (p. 259).
Si $D \vdash A : -$, en appliquant l'hypothèse de récurrence à D, on obtient $A.\mathrm{I} = \bot$.
Si $D \vdash A : +$, en appliquant l'hypothèse de récurrence à D, on obtient $A.\mathrm{I} = \bot$.
Voyons la seconde implication.
Supposons $\langle\downarrow (C)\rangle^\bot \leq C$, ce qui implique par définition du sous-typage, $D = \langle\downarrow (D)\rangle^\bot$.
Supposons aussi $C \vdash A : -$.
Nécessairement, $A \neq C$, et on peut appliquer la proposition 3.2.24 (p. 259).
Si $D \vdash A : +$, en appliquant l'hypothèse de récurrence à D, on obtient $A.\mathrm{I} = \bot$.
Si $C_2 \vdash A : -$, en appliquant l'hypothèse de récurrence à D, on obtient $A.\mathrm{I} = \bot$.
Montrons maintenant simultanément les deux implications de droite à gauche, toujours par récurrence sur C.

- $C = \mathrm{Unit}$

On a $C = \langle\downarrow (C)\rangle^{\perp}$.
• $C = C_1 \xrightarrow{\alpha} C_2$
Voyons la première implication.
Supposons $\forall A \neq \text{Unit}\,.\ C \vdash A : + \Rightarrow A.\text{I} = \perp$.
Pour montrer que $C \leq \langle\downarrow (C)\rangle^{\perp}$, il est suffisant de montrer que $\alpha = \perp$, $C_2 \leq \langle\downarrow (C_2)\rangle^{\perp}$ et $\langle\downarrow (C_1)\rangle^{\perp} \leq C_1$.
Comme $C \vdash C : +$, on déduit de l'hypothèse que $\alpha = \perp$.
Considérons un type A différent de Unit. Nous allons montrer que si $C_2 \vdash A : +$ est valide, alors $A.\text{I} = \perp$, de même que si $C_1 \vdash A : -$ est valide, alors $A.\text{I} = \perp$, ce qui permettra de conclure en appliquant l'hypothèse de récurrence à C_2 et C_1 respectivement.
Supposons $C_2 \vdash A : +$. Par la proposition 3.2.24 (p. 259), on a $C_1 \xrightarrow{\alpha} C_2 \vdash A : +$, donc par hypothèse $A.\text{I} = \perp$.
Supposons $C_1 \vdash A : -$. Par la proposition 3.2.24 (p. 259), on a $C_1 \xrightarrow{\alpha} C_2 \vdash A : +$, donc par hypothèse $A.\text{I} = \perp$.
On a donc montré $C \leq \langle\downarrow (C)\rangle^{\perp}$.
Voyons la seconde implication.
Supposons $\forall A \neq \text{Unit}\,.\ C \vdash A : - \Rightarrow A.\text{I} = \perp$.
Pour montrer que $\langle\downarrow (C)\rangle^{\perp} \leq C$, il est suffisant de montrer que $C_1 \leq \langle\downarrow (C_1)\rangle^{\perp}$ et $\langle\downarrow (C_2)\rangle^{\perp} \leq C_2$.
Considérons un type A différent de Unit. Nous allons montrer que si $C_2 \vdash A : -$ est valide, alors $A.\text{I} = \perp$, de même que si $C_1 \vdash A : +$ est valide, alors $A.\text{I} = \perp$, ce qui permettra de conclure en appliquant l'hypothèse de récurrence à C_2 et C_1 respectivement.
Supposons $C_2 \vdash A : -$. Par la proposition 3.2.24 (p. 259), on a $C_1 \xrightarrow{\alpha} C_2 \vdash A : -$, donc par hypothèse $A.\text{I} = \perp$.
Supposons $C_1 \vdash A : +$. Par la proposition 3.2.24 (p. 259), on a $C_1 \xrightarrow{\alpha} C_2 \vdash A : -$, donc par hypothèse $A.\text{I} = \perp$.
On a donc montré $\langle\downarrow (C)\rangle^{\perp} \leq C$.
• $C = \text{Ref}^{\alpha}(D)$
Voyons la première implication.
Supposons $\forall A \neq \text{Unit}\,.\ C \vdash A : + \Rightarrow A.\text{I} = \perp$.
Pour montrer que $C \leq \langle\downarrow (C)\rangle^{\perp}$, il est suffisant de montrer que $\alpha = \perp$ et $D = \langle\downarrow (D)\rangle^{\perp}$.
Comme $C \vdash C : +$, on déduit de l'hypothèse que $\alpha = \perp$.
Considérons un type A différent de Unit. Nous allons montrer que si $D \vdash A : +$ est valide, alors $A.\text{I} = \perp$, de même que si $D \vdash A : -$ est valide, alors $A.\text{I} = \perp$, ce qui permettra de conclure en appliquant l'hypothèse de récurrence à D.
Supposons $D \vdash A : +$. Par la proposition 3.2.24 (p. 259), on a $\text{Ref}^{\alpha}(D) \vdash$

$A : +$, donc par hypothèse $A.\mathrm{I} = \bot$.
Supposons $D \vdash A : -$. Par la proposition 3.2.24 (p. 259), on a $\mathrm{Ref}^{\alpha}(D) \vdash A : +$, donc par hypothèse $A.\mathrm{I} = \bot$.
On a donc montré $C \leq \langle\downarrow (C)\rangle^{\bot}$.
Voyons la seconde implication.
Supposons $\forall A \neq \mathrm{Unit}\,.\ C \vdash A : - \Rightarrow A.\mathrm{I} = \bot$.
Pour montrer que $\langle\downarrow (C)\rangle^{\bot} \leq C$, il est suffisant de montrer que $D = \langle\downarrow (D)\rangle^{\bot}$.
Considérons un type A différent de Unit. Nous allons montrer que si $D \vdash A : -$ est valide, alors $A.\mathrm{I} = \bot$, de même que si $D \vdash A : +$ est valide, alors $A.\mathrm{I} = \bot$, ce qui permettra de conclure en appliquant l'hypothèse de récurrence à D.
Supposons $D \vdash A : -$. Par la proposition 3.2.24 (p. 259), on a $\mathrm{Ref}^{\alpha}(D) \vdash A : -$, donc par hypothèse $A.\mathrm{I} = \bot$.
Supposons $D \vdash A : +$. Par la proposition 3.2.24 (p. 259), on a $\mathrm{Ref}^{\alpha}(D) \vdash A : -$, donc par hypothèse $A.\mathrm{I} = \bot$.
On a donc montré $\langle\downarrow (C)\rangle^{\bot} \leq C$.
⌞

Cette dernière proposition permet de définir un critère assurant qu'un type sécuritaire pour l'environnement est sûr. On dit qu'un type sécuritaire L_S est sûr si pour tout programme mobile typé $\lambda x : L.\,\mathsf{M}[x]$ (L étant égal à $\downarrow (L_S)$) et tout environnement instrumenté $\mathsf{L_S}$ de type sécuritaire L_S, l'exécution de

$$@^{(\langle M\rangle^{\bot},\bot)}(\langle\!\langle \lambda x : L.\,\mathsf{M}[x]\rangle\!\rangle^{\bot}, \mathsf{L_S})$$

est sûre, autrement dit n'entraîne pas la destruction par un destructeur étiqueté par $\bot$ d'une valeur étiquetée par $\top$. Il vient que le type sécuritaire L_S est sûr si tout type sortant de L_S et différent de Unit a pour étiquette $\bot$. La réciproque pourrait être montrée en adaptant le théorème 3.2.23 (p. 258) au cas des types sécuritaires. Ce théorème adapté permettrait alors d'associer à tout type sortant A de L_S un environnement de type L_S à partir duquel le type A serait accessible. Ainsi, s'il existait un type sortant A de L_S, différent de Unit et ayant pour étiquette $\top$, on obtiendrait une destruction proscrite pour un certain programme mobile appelant l'environnement local associé à A par ce théorème.

En résumé, dans le cas du code mobile, nous proposons de contrôler les accès aux ressources

– en vérifiant que l'instrumentation du code local est correctement réalisée, ce qui peut se faire en montrant que l'environnement local admet un type sécuritaire dans le système de types décrit dans la table 3.13

(p. 269),

– en vérifiant une forme de confinement des types à protéger, précisément en vérifiant qu'un type à protéger n'apparaît dans le type de l'environnement ni en une occurrence positive ni sous la garde d'un constructeur $\mathrm{Ref}^{\alpha}(-)$, ce qui permet de convertir le type sécuritaire de l'environnement pour confiner les types à protéger.

Conclusion

Nous avons étudié dans cette thèse l'exécution de code mobile dans un environnement local ; le code mobile étant potentiellement hostile, l'environnement local est chargé de protéger les ressources du système hôte, précisément de garantir d'une part, la confidentialité des informations contenues dans les ressources, d'autre part, le confinement local de ressources à protéger. Au terme de ce travail, nous pouvons affirmer qu'il est possible d'analyser l'environnement local seul pour garantir la sécurité de l'exécution de tout code mobile. Nous avons pu définir deux critères de sécurité :

- le premier est vérifié par le code de l'environnement local si et seulement si l'environnement garantit la confidentialité des ressources locales,
- le second porte sur le type de l'environnement local et implique que les ressources à protéger sont confinées dans l'environnement si le type de l'environnement le vérifie.

L'originalité de ces résultats ne tient pas en la possibilité d'analyser un programme pour déterminer des propriétés concernant le contrôle des flux d'informations ou des accès : la littérature sur ces sujets est déjà vaste. Mais elle provient de la problématique propre à l'exécution de code mobile, en particulier des deux points spécifiques suivants :

- seul l'environnement local est disponible pour l'analyse,
- un critère de sécurité doit être valide pour tout code mobile envisageable.

Rentrons un peu plus dans le détail de ces critères pour en cerner les limites et étudier les extensions possibles. Noter que nous ne revenons pas sur les limites impliquées par nos hypothèses de travail et décrites en introduction (cf. p. 7) : la principale hypothèse concerne l'utilisation d'un langage de haut niveau, ce qui suppose qu'on s'assure de la préservation par compilation des propriétés de sécurité.

Nous comparons les deux critères suivant trois points de vue :

- le langage de programmation utilisé,

– le degré d'approximation du critère relativement à la propriété qu'il implique,
– les techniques utilisées pour démontrer ces critères.

Commençons par une technique commune aux deux démarches. Tant la confidentialité que le confinement ont été étudiés à partir d'une annotation du langage de programmation, où chaque opérateur du langage reçoit une étiquette : dans les deux cas, ce sont des langages bien connus, qui peuvent être considérés comme les langages naturels pour l'étude de ces questions.
Excepté cette technique commune, le contraste est autrement parfait. Pour le critère de confidentialité, le langage est élémentaire, le critère est exact, c'est-à-dire équivalent à la propriété de confidentialité, et les techniques sont élaborées, puisqu'une partie des deux premiers chapitres est dévolue à leur développement ; pour le critère de confinement, le langage est enrichi (par des références), le critère repose sur l'approximation des valeurs par leur type, et les techniques sont élémentaires.

Pour la confidentialité, c'est donc surtout une extension du langage de programmation qui est souhaitable. Si on reprend la même démarche, il conviendrait pour le langage étendu de

1. définir une sémantique observationnelle,
2. étudier l'uniformité et la dépendance, en recourant à une interprétation abstraite,
3. donner un critère de confidentialité, à partir de l'observation abstraite.

Des trois points, le premier semble le plus délicat. Depuis les travaux de Moggi [58], il est reconnu que l'expression d'une sémantique dénotationnelle peut être paramétrée par une monade. D'un point de vue opérationnel, la monade décrit la structure des configurations de la machine abstraite exécutant les programmes. Dans l'optique d'une généralisation, il serait donc intéressant de combiner l'approche monadique et l'approche observationnelle. Il s'avère que la méthode de Howe a déjà été utilisée pour des monades particulières. Intéressons-nous à la monade permettant de représenter un état-mémoire (global), adaptée au langage avec références utilisé pour le confinement. Dans [72], Ritter et Pitts montrent que la relation de bisimilarité est incluse strictement dans la relation d'équivalence contextuelle : c'est un résultat d'adéquation. On peut considérer que la relation de bisimilarité qu'ils définissent s'obtient à partir d'une notion d'observation : l'observation d'un programme convergent est obtenu en observant de son résultat à la fois la valeur obtenue et l'état-mémoire. Comme l'équivalence contextuelle est définie uniquement en observant du résultat la convergence, elle est clairement plus grossière que la bisimilarité : il faudrait raffiner l'observation du

résultat de manière à prendre en compte l'état-mémoire pour espérer obtenir la bisimilarité. À l'inverse, si on laisse inchangée l'équivalence contextuelle, c'est la notion d'observation utilisée pour définir la bisimilarité qui doit être moins discriminante. Lors de l'observation d'un programme, il est impossible d'observer le contenu d'une référence créée par le programme si elle n'a pas été rendue accessible par le programme. Autrement dit, pour un état-mémoire, il est impossible d'observer les références confinées dans le programme. C'est cette distinction qu'exploitent Jeffrey et Rathke dans [40, §4, 5] pour aboutir à l'adéquation complète, soit l'égalité entre la bisimilarité et l'équivalence contextuelle ; ils utilisent également la méthode de Howe, qui justifie ainsi par sa robustesse son emploi dans ce travail.

Pour le confinement, le langage pourrait être étendu par des structures de données, sans modifier l'étude de la frontière telle que nous l'avons menée : par exemple, au niveau des types, outre des types de base, comme des entiers ou des booléens, on pourrait introduire le produit cartésien ou la somme (union disjointe) sans aucune difficulté.
Comme le langage contient déjà des références, il se rapprocherait alors d'un langage orienté objets, comme Java. Un type dans un langage comme Java, même s'il est d'abord le nom d'une classe ou d'une interface, peut aussi être identifié avec la liste des signatures[13] de ses méthodes ; comme il est susceptible d'apparaître dans le type d'une de ses méthodes, il peut former un type récursif. Que devient l'étude de la frontière en présence de types récursifs ? Il apparaît que notre étude ne nécessite aucune hypothèse concernant le caractère non récursif des types. Cette dernière hypothèse n'est utilisée qu'une seule fois, pour associer à chaque type un programme convergent de ce type, lorsqu'on montre que le critère de confinement est optimal.
Autre particularité d'un système de types comme celui du langage Java : le sous-typage. La relation de sous-typage permet d'affirmer qu'un type $(m_i : A_i)_{i\in I}$, où pour chaque indice i de l'ensemble fini I, m_i est le nom d'une méthode et A_i son type, est un sous-type de $(m_j : A_j)_{j\in J}$, où J est inclus dans I. Avec le sous-typage, sans hypothèses sur la conversion des types, il n'est pas raisonnable de définir un critère de confinement à partir du type de l'environnement local. En effet, s'il est possible de réaliser une coercition d'un type vers un sous-type, c'est non seulement le type de l'environnement local qui doit vérifier le critère de confinement, mais aussi tous ses sous-types. Si comme en Java, il existe un type au sommet de la hiérarchie de sous-typage (le type « Object » en Java), le critère peut devenir inapplicable. Il est donc souhaitable d'interdire ou bien la conversion des

[13] La signature d'une méthode contient le nom et le type de la méthode.

types devant être confinés en sur-types, ou bien la coercition. Ce sont des hypothèses habituelles pour le confinement, la première donnée par exemple par Bokowski et Vitek dans [83, §5.3].
Une autre extension possible concernerait l'introduction du polymorphisme paramétrique : c'est la possibilité de paramétrer par un type la définition d'un programme. Le passage d'un système de types monomorphes à un système de types polymorphes impose de nombreuses modifications techniques à notre étude de la frontière, et demande donc d'être exploré plus avant. Il convient aussi de remarquer que le polymorphisme paramétrique permet de rendre abstrait dans le code mobile un type à protéger : il suffit en effet de paramétrer le code mobile par ce type pour empêcher tout accès à une valeur de ce type au sein du code mobile. Cette technique de confinement est étudiée par Leroy et Rouaix dans [49, §5.2].
S'il est possible d'envisager un critère de confinement pour un langage étendu, il reste que ce critère repose sur une approximation forte : il concerne les types et non les valeurs. C'est dommageable, dans la mesure où le processus d'approximation n'est pas formalisé. Il serait préférable de définir un critère de confinement exact, à partir du code de l'environnement local, équivalent à la propriété de confinement, puis de déduire par une interprétation abstraite transformant chaque valeur en son type le critère approché que nous donnons. Il s'agirait donc de déterminer l'ensemble des valeurs accessibles par le code mobile à partir de l'environnement local : on remarque un intéressant point de convergence avec le problème de l'adéquation complète en présence de références, évoqué plus haut.

Évoquons enfin brièvement les techniques mathématiques décrites dans le premier chapitre et utilisées dans les chapitres suivants. Elles concernent des objets infinis, au sens de structures arborescentes infinies en profondeur. Rappelons que pour définir ces objets infinis, nous avons privilégié une approche équationnelle, et que pour raisonner sur ces objets infinis, nous avons privilégié une approche déductive. À l'usage, ces deux approches se sont révélées efficaces et intuitives. Elles sont susceptibles d'être étendues dans de nombreuses directions. On pourrait par exemple considérer des systèmes d'équations récursives non déterministes, permettant de définir des ensembles d'arbres, en s'inspirant des schémas de programmes récursifs non déterministes développés par Arnold et Nivat dans [8] ; on pourrait aussi comparer l'approche déductive avec l'approche par points fixes pour d'autres théorèmes de points fixes que celui de Tarski.

Bibliographie

[1] Martin Abadi. Protection in programming-language translations. *Lecture Notes in Computer Science*, 1443 :868–883, 1998.

[2] Martin Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '99)*, pages 147–160. ACM Press, 1999.

[3] Martin Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 83–91, 1996.

[4] Samson Abramsky. The lazy lambda-calculus. In David Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[5] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2) :159–267, 1993.

[6] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter C.7, pages 739–782. North-Holland, 1977.

[7] Peter Aczel. *Non-Well-Founded Sets*, volume 14 of *CSLI Lecture Notes*. CSLI Publications, Stanford, California, 1988.

[8] André Arnold and Maurice Nivat. Non deterministic recursive program schemes. In Marek Karpiński, editor, *Proceedings of the 1977 International Conference on Fundamentals of Computation Theory*, volume 56 of *Lecture Notes in Computer Science*, pages 12–21. Springer-Verlag, 1977.

[9] Jos Baeten and W. Peter Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[10] Anindya Banerjee and David Naumann. Representation independence, confinement and access control. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '02)*, volume 37(1) of *ACM SIGPLAN Notices*, pages 166–177. ACM Press, 2002.

[11] Henk Barendregt. *The Lambda Calculus : its Syntax and Semantics*. North-Holland, 1981 (1st ed) revised 84.

[12] Henk Barendregt. The impact of the lambda calculus on logic and computer science. *Bulletin of Symbolic Logic*, 3(3) :181–215, 1997.

[13] Jon Barwise and Lawrence Moss. *Vicious Circles : On the Mathematics of Non-Wellfounded Phenomena*. CSLI Publications, Stanford, California, 1996.

[14] Gérard Berry. Some syntactic and categorical constructions of lambda calculus models. Rapport de Recherche 80, Institute National de Recherche en Informatique et en Automatique (INRIA), 1981.

[15] Frédéric Besson, Thomas de Grenier de Latour, and Thomas Jensen. Secure calling contexts for stack inspection. In *Proceedings of the 4th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '02)*, pages 76–87. ACM Press, 2002.

[16] Inge Bethke, Jan Willem Klop, and Roel de Vrijer. Descendants and origins in term rewriting. *Information and Computation*, 159 :59–124, 2000.

[17] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33 :309–338, 1998.

[18] Luca Cardelli. Type systems. In Allen Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[19] Ellis Cohen. Information transmission in sequential programs. In Richard DeMillo, David Dobkin, Anita Jones, and Richard Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, New York, 1978.

[20] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Selected Papers of the First International Workshop on Types for Proofs and Programs (TYPES '93)*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1994.

[21] Bruno Courcelle. Arbres infinis et systèmes d'équations. *R.A.I.R.O., Informatique Théorique*, 13 :31–48, 1979.

[22] Bruno Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13 :131–180, 1979.

[23] Bruno Courcelle. Recursive applicative program schemes. In van Leeuwen [82], chapter 9, pages 460–492.

[24] Patrick Cousot and Radhia Cousot. Inductive definitions, Semantics and Abstract Interpretation. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '92)*, pages 84–94. ACM Press, 1992.

[25] Nikolas de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34 :381–392, 1972.

[26] Dorothy Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5) :236–243, 1976.

[27] Sophia Drossopoulou and Susan Eisenbach. Java is type safe – probably. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 - Object-Oriented Programming, 11th European Conference, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[28] Milind Gandhe, G. Venkatesh, and Amitabha Sanyal. Labeled λ-calculus and a generalized notion of strictness. *Lecture Notes in Computer Science*, 1023 :103–110, 1995.

[29] Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy (S& P '82)*, pages 11–20. IEEE Computer Society Press, 1982.

[30] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox : An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems (USITS '97)*, pages 103–112. USENIX, 1997.

[31] Li Gong and Roland Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the 1998 Network and Distributed System Security Symposiums (NDSS '98)*, pages 125–134. Internet Society, 1998.

[32] Andrew Gordon and Andrew Pitts, editors. *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1998.

[33] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4) :201–206, 1994.

[34] Nevin Heintze and Jon Riecke. The SLam calculus : Programming with security and integrity. In POPL '98 [69], pages 365–377.

[35] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15 :81–91, 1950.

[36] Furio Honsell and Marina Lenisa. Final semantics for untyped lambda-calculus. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications (TLCA '95)*, volume 902 of *Lecture Notes in Computer Science*, pages 249–265. Springer-Verlag, 1995.

[37] Douglas Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2) :103–112, 1996.

[38] J. Martin E. Hyland and C.-H. Luke Ong. On full abstraction for PCF : I, II, and III. *Information and Computation*, 163(2) :285–408, 2000.

[39] Husain Ibraheem and David Schmidt. Adapting big-step semantics to small-step style : Coinductive interpretations and "higher-order" derivations. In Andrew Gordon, Andrew Pitts, and Carolyn Talcott, editors, *Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS '97)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[40] Alan Jeffrey and Julian Rathke. Towards a theory of bisimulation for local names. In *Fourteenth Annual Symposium on Logic in Computer Science (LICS '99)*, pages 56–66. IEEE Computer Society Press, 1999.

[41] Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Verification of control flow based security properties. In *Proceedings of the IEEE Symposium on Security and Privacy (S& P '99)*, pages 89–105. IEEE Computer Society Press, 1999.

[42] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS '87)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[43] Akihiro Kanamori. The mathematical import of Zermelo's Well-Ordering Theorem. *Bulletin of Symbolic Logic*, 3(3) :281–311, 1997.

[44] Peter Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4) :308–320, 1964.

[45] Peter Landin. A lambda-calculus approach. In *Advances in Programming and Non-Numerical Computation*, pages 97–141, Oxford, 1966. Pergammon Press.

[46] Xavier Leroy. Polymorphism by name for references and continuations. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '93)*, pages 220–231. ACM Press, 1993.

[47] Xavier Leroy. Java bytecode verification : An overview. *Lecture Notes in Computer Science*, 2102 :265–285, 2001.

[48] Xavier Leroy and Atsushi Ohori, editors. *Proceedings of the Second International Workshop on Types in Compilation (TIC '98)*, volume 1473 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[49] Xavier Leroy and François Rouaix. Security properties of typed applets. In POPL '98 [69], pages 391–403.

[50] Jean-Jacques Lévy. An algebraic interpretation of the λ-β-K-calculus and a labelled λ-calculus. In Corrado Böhm, editor, *λ-calculus and Computer Science Theory, Proceedings of the Symposium*, volume 37 of *Lecture Notes in Computer Science*, pages 147–165, 1975.

[51] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6) :1811–1841, November 1994.

[52] Ian Mason, Scott Smith, and Carolyn Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1) :26–47, 1996.

[53] Robin Milner. Fully abstract models of typed lambda-calculus. *Theoretical Computer Science*, 4 :1–22, 1977.

[54] Robin Milner. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[55] Robin Milner. Operational and algebraic semantics of concurrent processes. In van Leeuwen [82], chapter 19, pages 1201–1242.

[56] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1) :209–220, 1991.

[57] John Mitchell. Type systems for programming languages. In van Leeuwen [82], chapter 8, pages 366–458.

[58] Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. of 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89*, pages 14–23. IEEE Computer Society Press, 1989.

[59] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In Leroy and Ohori [48], pages 28–52.

[60] George Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '97)*. ACM Press, 1997.

[61] C.-H. Luke Ong. Correspondence between operational and denotational semantics. In Samson Abramsky, Dov Gabbay, and Tom Maibaum, editors, *Handbook of Logic in Computer Science, Vol 4*, pages 269–356. Oxford University Press, 1995.

[62] David Park. Fixpoint induction and proofs of program properties. In *Machine Intelligence*, volume 5, pages 59–78. Edinburgh University Press, 1969.

[63] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science : 5th GI-Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[64] Lawrence Paulson. Set theory for verification II : Induction and recursion. *Journal of Automated Reasoning*, 15(2) :167–215, 1995.

[65] Andrew Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124(2) :195–219, 1994.

[66] Gordon Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1 :125–159, 1975.

[67] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3) :223–255, 1977.

[68] Gordon Plotkin. A structural approach to operational semantics. Research report DAIMI FN-19, Aarhus University, 1981.

[69] *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '98)*. ACM Press, 1998.

[70] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, 2000.

[71] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Programming Languages and Systems : 10th European Symposium on Programming, ESOP 2001, Proceedings*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer-Verlag, 2001.

[72] Eike Ritter and Andrew Pitts. A fully abstract translation between a λ-calculus with reference types and Standard ML. In *2nd International Conference on Typed Lambda Calculus and Applications (TLCA '95)*, volume 902 of *Lecture Notes in Computer Science*, pages 397–413. Springer-Verlag, 1995.

[73] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal of Selected Areas in Communications*, 21(1) :5–19, 2003.

[74] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1) :59–91, 2001.

[75] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5) :447–479, 1998.

[76] David Schmidt. Trace-based abstract interpretation of operational semantics. *Lisp and Symbolic Computation*, 10(3) :237–271, 1998.

[77] Dana Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1–2) :411–440, 1993 (Reprint of a manuscript written in 1969).

[78] Christian Skalka and Scott Smith. Static use-based object confinement. In Iliano Cervesato, editor, *Proceedings - Foundations of Computer Security (FCS '02)*, volume 02-12 of *DIKU technical reports*, pages 117–126, 2002.

[79] David Sutherland. A model of information. In *Proceedings of 9th National Computer Security Conference*, pages 175–183, Gaithersburg, Md., 1986.

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

[81] Mads Tofte. Operational semantics and polymorphic type inference. Technical Report EDC-LFCS-88-54, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, 1987. PhD Thesis.

[82] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science – Volume B : Formal Models and Semantics*. Elsevier, MIT Press, 1990.

[83] Jan Vitek and Boris Bokowski. Confined types. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 82–96. ACM Press, 1999.

[84] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3) :167–187, 1996.

[85] Dan Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, Department of Computer Science, 1999.

[86] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1) :38–94, 1994.

Index