Thèse

présentée à

L'Université Paris VI

pour obtenir le titre de

Docteur de l'Université Paris VI

Spécialité :

Informatique

par

Jean-Pierre TALPIN

Sujet de thèse :

Aspects Théoriques et Pratiques de l'Inférence de Type et d'Effets

Soutenue le 12 Mai 1993 devant le jury composé de :

MM.	Jean	BERSTEL	Président
	Michel	MAUNY	Rapporteurs
	Mads	TOFTE	
	Alessandro	GIACALONE	Examinateurs
	Pierre	JOUVELOT	
	Christian	QUEINNEC	

Thèse préparée à :

l'Ecole Nationale Supérieure des Mines de Paris

Rapport EMP/CRI/A-236

Remerciements

Une thèse commence traditionellement par les remerciement du candidat aux membres de son jury, à ses collègues, à ses amis et à ses proches. Il est tout aussi usuel que le candidat les rédige en dernier, laissant, volontairement ou non, transparaitre l'état inhabituel sinon l'émotion qui y succède. Je n'y faillirai pas!

Sans plus attendre, je remercie Jean Berstel, Professeur à l'Université Paris 6, d'avoir bien voulu présider mon jury.

Michel Mauny, Chargé de Recherche a l'INRIA, a accepté d'être mon rapporteur et je l'en remercie chaleureusement.

Je remercie également Mads Tofte, Professeur à DIKU (l'Institut d'informatique de l'Université de Copenhague) d'avoir accepté de rapporter cette thèse. Je le remercie également, lui et le Professeur Neil Jones, pour m'avoir invité à l'Université de Copenhague à plusieurs reprises.

Alessandro Giacalone, Chef de Projet à l'E.C.R.C. (European Computer-Industry Research Center), m'accueillera cet été pour travailler dans son équipe. Il a souhaité faire partie de mon jury de thèse et je lui en suis très reconnaissant.

Pierre Jouvelot, Chef de Projet au C.R.I. (Centre de Recherche en Informatique), a étroitement encadré mon travail tout au long de ma thèse. J'ai beaucoup apprécié la méthode, la rigueur et l'honnèteté qui le caractérisent dans son travail. Il partage tous les résultats publiés de cette thèse et je le remercie chaleureusement de sa contribution.

Christian Queinnec, Chef de Projet à l'INRIA, m'a soutenu dans les débuts difficiles de cette thèse, m'accueillant dans son équipe à l'Ecole Polytechnique durant mon service national, puis acceptant d'être mon directeur de thèse à Paris 6. Je lui en suis très reconnaissant.

Michel Lenci, Directeur de l'I.S.I.A. (Institut Supérieur d'Informatique et d'Automatique), m'a accueilli au C.R.I. qu'il dirigeait alors. Je lui en suis reconnaissant et je garde de lui le souvenir d'un directeur unique par sa gentillesse et sa symphatie.

Je remercie également Alain Deutsch, Vincent Dornic, Dannie Durand, Phillipe Bazet et Francois Masdupuy qui ont été mes collègues thésards, pour leur aide et leurs marques de sympathie et d'amitié.

Enfin, je remercie Corinne Ancourt, François Irigoin, Jacqueline Altimira, Annie Pech-Ripaud et les autres membres du C.R.I. pour leur sympathie. En particulier, je tiens à dire un grand merci à Kathrin MacKinley, pour son aide trés précieuse dans la correction de mes nombreuses fautes d'anglais.

Enfin je dédie cette thèse à mon futur enfant et à sa maman, Yan-Mei, à mes parents, Annie et Maurice, et à mon frère, Eric.

Contents

Introduction

Dans de nombreuses disciplines scientifiques, comme en informatique, et tout spécialement dans le domaine des langages de programmation, il est communément acquis d'associer aux études théoriques, sur les modèles de calcul, des travaux de nature plus pratique de mise en oeuvre, ou d'implémentation.

Pour cela, il est souhaitable que les techniques de mise en œuvre des langages reposent sur des méthodes formelles qui permettent d'en assurer la correction. Ce souci de rigueur ne s'oppose pas aux impératifs de confort d'utilisation et d'efficacité des outils de programmation. Bien au contraire, rigueur et preuve de correction sont autant de gages quant à l'universalité de l'outil de programmation et la sécurité de son emploi par rapport à ses spécifications. Toutes deux sont bénéfiques à sa bonne maîtrise par l'utilisateur, pour aboutir finalement à une productivité et une qualité de développement améliorées.

L'usage de méthodes formelles appropriées permet d'enrichir notablement le pouvoir d'expression d'un langage de programmation. Ainsi, l'analyse sémantique, dont l'objet est de déterminer des propriétés formelles de programmes, procure une aide notable à la vérification, la documentation et l'optimisation de programmes.

Typage Statique

Parmi ces méthodes formelles, le *typage statique* est sans doute la forme la plus populaire. Le typage statique consiste à détecter au moment de la compilation une source fréquente d'erreurs d'exécution de programmes: l'usage incohérent d'une valeur par rapport à la structure de cette valeur (comme par exemple ajouter `1` à la valeur booléenne `true`). La vérification statique de typage s'effectue en associant un type à chacune des valeurs manipulées par un programme et en vérifiant que l'utilisation des valeurs est conforme aux types affectés à celles-ci.

Dès lors que le typage statique d'un programme est vérifié, aucune erreur d'accès aux données ne peut se produire pendant son exécution. Mais parce que les règles décrivant le typage statique des programmes se doivent d'être simples pour être comprises par les utilisateurs, et parce que les programmes peuvent pourtant être complexes, la faiblesse inhérente à tout système de typage statique est de ne pouvoir accepter qu'une partie des programmes dont le sens est correct et de devoir rejeter certains d'entre eux.

Dans la recherche d'un meilleur compromis entre simplicité et performance, l'introduction de la notion de *polymorphisme* a été à l'origine de progrès notables, en permettant le typage statique des fonctions génériques. Les fonctions génériques, comme par exemple la fonction identité, peuvent opérer sur des données de structures différentes. L'intérêt pour les fonctions génériques tient au fait que celles-ci peuvent être codées et compilées une fois pour toutes puis être utilisées dans de nombreuses situations sans aucune modification.

Sans la notion de polymorphisme, le typage statique ne permet pas l'utilisation des fonctions génériques. C'est le cas en Pascal, ou en Fortran. Pour contourner cette interdiction, le programmeur a parfois recours à certaines astuces, en dupliquant le code d'une fonction pour autant de types différents que de contextes d'utilisation. Dans d'autres langages de programmation, comme Ada, cela peut être fait de manière plus rigoureuse: l'utilisateur peut définir explicitement des fonctions génériques que le compilateur se chargera de dupliquer avant de compiler.

A l'instar des systèmes de typage tels que ceux de Pascal, de Fortran ou de Ada, le typage polymorphe supporte naturellement la notion de fonctions génériques et permet leur réutilisation. C'est un avantage notable car, en pratique, les fonctions génériques sont très utiles dès lors qu'il est besoin, dans un programme, de manipuler des listes, des arbres ou des graphes.

L'intérêt pour le typage statique a motivé l'élaboration des techniques qui ont permis d'automatiser ce procédé. Tout d'abord, [Morris, 1968] mit en évidence la relation entre la structure d'un programme et son typage au moyen d'équations linéaires récursives. Ensuite, [Hindley, 1969] développa une technique pour déterminer le type de lambda termes par une méthode de résolution utilisant un algorithme d'unification [Robinson, 1965]. Fort de ces résultats, [Milner, 1978] introduisit l'inférence de type polymorphe, combinant les techniques d'inférence de type avec une notion de polymorphisme. Ce fut là un progrès notable qui a, depuis, été le sujet de nombreux travaux et extensions.

Dans [Milner, 1978], où l'auteur introduit le langage ML, le polymorphisme s'exprime au niveau de la construction syntaxique de liaison lexicale: le `let`. La manière la plus simple de considérer la forme `let` de ML est de la considérer comme une abréviation: l'expression `(let (x e) e'`) a le même sens que la substitution de `x` par `e` dans `e'`, notée `e'[e/x]`. Il en va de même pour le type de cette expression: lorsque l'on type l'expression `e'[e/x]`, chaque occurrence de `e` peut avoir un type différent des autres. Dans la discipline de typage de Milner, cela s'exprime en associant un schéma de type à `x`. Le schéma de type de `x` permet de représenter l'ensemble des types possibles de l'expression `e` pour chaque occurence de `x` dans `e'`.

Langages Impératifs et Disciplines de Typage

Le typage polymorphe est, on le voit, parfaitement approprié pour les langages fonctionnels. Qu'en est-il pour les langages dit impératifs ? Ceux-ci sont, habituellement, définis en opposition aux langages fonctionnels, tels que ML, parce qu'ils supportent d'autres opérations, permettant notamment l'affectation, c'est-à-dire la modification *en place* d'une structure de donnée (comme la valeur d'un pointeur, par exemple).

Mais l'ajout de traits de programmation impérative à un langage, pour permettre par exemple la manipulation de pointeurs, s'accompagne de la nécessité d'introduire une notion d'*état* pour comprendre le sens des programmes. Cela rend tout de suite invalide notre précédente définition de l'expression `let` comme abbréviation: l'expression `(let (x e) e'`) n'a plus le même sens que `e'[e/x]` et l'extension de la notion de polymorphisme à la ML aux opérations sur les pointeurs apparaît d'ores et déjà suspecte.

Pour typer les opérations sur les pointeurs en ML, on peut penser à définir le type $ref(\tau)$ pour représenter l'ensemble des pointeurs référençant une valeur de type τ. Alors, l'opération d'initialisation de pointeurs `new`, qui accepte toute valeur de type τ et retourne un pointeur vers celle-ci, devrait avoir le type $\tau \rightarrow ref(\tau)$.

Pourtant, une fois initialisé, un pointeur doit toujours être associé au même type. Autrement, d'aucuns pourraient lui associer une valeur d'un type donné, *int* par exemple, puis lire cette valeur et prétendre qu'elle ai un type différent, pourquoi pas *bool*. Or, ajouter `1` à `true` est justement une situation que l'on cherche à éviter à l'aide du typage statique. On le voit très vite, il semble apparaître une incompatibilité flagrante entre des opérations de style impératif et le typage polymorphe.

Comme nous le voyons, l'ajout de traits impératifs à un langage tel que ML modifie le sens de la construction `let`. Donc, la manière de typer cette expressions doit également changer. Il convient de définir une discipline de typage tenant compte des traits impératifs du langage: une discipline de typage impérative. Il ressort de notre précédente discussion que le type d'un pointeur ne peut pas être généralisé par un schéma de type comme l'était toute autre valeur. Tout système de typage polymorphe supportant le typage d'opérations impératives doit donc, à un niveau ou un autre, prendre en compte, de manière approximative, la tranformation d'état provoquée par de telles opérations.

La technique la plus classique pour typer correctement les traits impératifs en ML est la discipline de typage impérative [Tofte, 1987, Tofte, 1990] et son extension basée sur les "types faibles", aujourd'hui utilisée dans les différentes mises en oeuvre du langage Standard ML [Milner & al., 1990, Appel & Mac Queen, 1990]. Une autre approche, suggérée par [Leroy, 1990], consiste à marquer le type des fonctions avec l'ensemble des types associés aux variables libres de ces fonctions: c'est le "typage des fermetures". L'idée maîtresse est ici d'interdire ensuite la généralisation de types "dangereux": le type des pointeurs apparaissant dans le type des fonctions.

Toutes ces approches sont bien évidemment approximatives quant à réellement représenter la transformation d'état que provoque une opération impérative. En pratique, la plupart d'entre elles ne permettent pas le typage correct de fonctions d'ordre supérieur en présence de traits impératifs. D'autres sont même parfois incapables de typer certaines expressions qui ne mettent pourtant en œuvre aucune opération impérative.

Systèmes d'Effet et Disciplines de Typage

A l'instar de ces tentatives, les systèmes d'effets en général, et le langage de programmation FX en particulier, permettent d'intègrer le typage polymorphe et la programmation impérative. Indépendamment, d'autres d'investigations ont également œuvré à l'intégration d'une notion de transformation d'état au typage polymorphe d'opérations impératives [Damas, 1985, O'Toole, 1990, Wright, 1992].

Le langage FX est fortement inspiré de Scheme [Rees & al., 1988] et de ML. Il supporte les fonctions d'ordre supérieur, des constructions impératives, l'allocation dynamique et la récupération automatique des structures de donnée, mais surtout un système de compilation séparée [Sheldon & Gifford, 1990] et un typage statique polymorphe.

Plus encore, il utilise un système d'effets. De même qu'un type représente ce qu'un programme calcule, un effet décrit comment ce programme calcule. Types et effets sont annotés par des *régions*. Les régions décrivent des relations de partage entre les zones mémoire où résident les structures de données. Doté d'une telle richesse d'expression, ce langage permet à son utilisateur de s'exprimer tout aussi naturellement dans un style de programmation fonctionnel que dans un style impératif, forgé à l'usage de langages plus traditionnels comme Pascal ou Fortran.

Dans [Gifford & al., 1987], les auteurs présentent la sémantique statique du langage FX par un système pour la vérification de déclarations de types et d'effets polymorphes. Le besoin de spécifier type, région et effet de chaque fonction d'un programme est un travail assez ennuyeux. Pour remédier à ce problème, les auteurs présentent dans [Jouvelot & Gifford, 1991] le premier système permettant d'inférer automatiquement des effets sur le principe dit de "reconstruction algébrique": ils y définissent le problème de l'inférence d'effet par analogie directe avec le principe d'inférence de type par résolution de contraintes de [Morris, 1968].

Contribution

Nous présentons ici un système de typage statique qui est basé sur les notions de système d'effet et d'inférence de type polymorphe, et qui, à l'instar de la reconstruction algébrique, détermine statiquement le type et l'effet principal des programmes. Comme l'explique le chapitre 2, cela est possible par l'adjonction à l'ensemble des règles de typage statique d'une règle explicitant une relation d'ordre entre les effets des expressions. Cette relation d'ordre est analogue à la notion de relation de sous-typage, mais elle porte sur les effets.

L'inférence de type et d'effet est ici étendue à l'inférence de régions. Les régions apparaissent dans les types. Par exemple, $ref_\rho(\tau)$ est un pointeur de type τ dans la région ρ. Elles apparaissent dans les effets. $init(\rho)$ est l'effet d'initialiser un pointeur dans la région ρ. Les régions décrivent statiquement les relations de partages entre les données que manipule un programme. Incidemment, elle permettent de localiser précisement la localité des données et de déterminer celles d'entre elles sur lesquelles portent les effets de bords.

Cela a, nous le verrons, des conséquences tant pratiques que théoriques et nous nous attachons à les développer formellement. Sur le plan pratique, le fait d'avoir statiquement une idée de la localité d'une donnée permet d'en optimiser la gestion: l'allocation et la récupération.

Sur un plan plus général, et afin d'intégrer des traits de programmation impérative dans un langage fonctionnel, mais avant tout de montrer combien il est aisé et naturel de le faire en utilisant un système d'effet, nous définissons une discipline de typage impérative qui est basée sur l'inférence d'effet. L'idée essentielle de ce système est d'utiliser l'inférence d'effet afin d'estimer au mieux la transition d'état mémoire qui est provoquée par les effets de bords d'un programme.

Pour typer les pointeurs, nous utilisons des effets de la forme $init(\rho,\tau)$, $read(\rho,\tau)$ et $write(\rho,\tau)$ qui informent du type τ des données pouvant être référencées par les pointeurs appartenant à la région ρ. Ces effets sont utilisés pour contrôler la généralisation du type des pointeurs. Pour typer une expression de la forme `(let (x e) e')`, il est suffisant de connaître les effets d'initialisation de l'expression `e` afin de savoir comment généraliser son type.

Pour ne rapporter que les effets d'initialisation afférant à des régions auxquelles appartiennent des pointeurs effectivement accessibles par le programme, nous définissons un critère *d'observation*. Les effets observables d'une expression sont exactement ceux qui portent sur une région ρ accessible, c'est à dire libre

dans l'environnement de typage de l'expression ou bien dans le type de sa valeur. Tous les autres effets sont associés à des régions ayant été allouées pour un usage temporaire et local et ne sont donc pas observables.

La notion d'effet observable est cruciale pour distinguer les fonctions qui utilisent des pointeurs localement et temporairement. Parmi de telles fonctions, on rencontre nombre qui ne sont que les versions impératives des fonctions usuellement définies dans un style de programmation purement fonctionnel. En utilisant l'inférence d'effet et muni du critère d'observation, notre système de typage est capable de déterminer très précisément la portée des effets de bord, nous assurant d'une généralisation de type plus efficace et plus uniforme qu'avec d'autres méthodes.

Motivations

L'intérêt porté aux langages de programmation fonctionnels tels que ML ne se limite pas au seul problème du typage polymorphe. Il est très certainement lié à la relation étroite de ces langages avec le lambda-calcul de [Barendregt, 1984] et la facilité de les définir au moyen de méthodes de sémantique formelle [Plotkin, 1981, Stoy, 1977]. Ce lien étroit entre les langages fonctionnels et le lambda-calcul facilite également l'utilisation de systèmes de preuve, comme [Coquand & Huet, 1988] et [Huet, 1989] ou bien [Harper & al., 1987], basés sur les logiques et lambda-calcul d'ordre supérieur [Girard, 1972, Girard, 1986], afin de démontrer la correction de techniques de compilation [Hannan, 1990, Hannan & Pfenning, 1992] et d'optimisation [Wand, 1991, Wand, 1992].

Ce critère de correction pourrait à moyen terme être source d'intérêts des milieux industriels pour les langages fonctionnels. Pour préparer ces outils à cet élargissement de l'audience des langages fonctionnels, il est nécessaire de proposer des langages plus sûrs, plus expressifs, aux performances encore plus importantes. Les fondements des langages fonctionnels, basés sur le lambda-calcul, permettent d'envisager cette perspective avec sérénité, puisqu'ils forment un excellent support pour formaliser et mettre en œuvre les techniques d'analyse, de compilation et d'optimisation à venir.

Un autre intérêt, du typage polymorphe cette fois, est de procurer des informations qui se révèlent utiles aussi bien pour le programmeur, qui peut par ce moyen décrire la spécification de ses applications, que pour le compilateur qui peut utiliser les informations de type pour produire un code plus efficace et une représentation de données moins coûteuse [Goldberg, 1991, Goldberg, 1992, Leroy, 1990].

L'inférence d'effets permet de déterminer les effets de bords et les relations de partage entre structures de données dans un programme. Ces informations peuvent également être utilisées par le compilateur, afin de déterminer la classe d'allocation des structures de données manipulées dans le programme: registre, pile ou tas [Tofte & Talpin, 1993, Talpin & Jouvelot, 1993, Tang & Jouvelot, 1992], ou bien encore pour effectuer des transformations pouvant, pourquoi pas, changer l'ordre d'évaluation des expressions du programme [Talpin & Jouvelot, 1993].

Implémentation

Lorsque l'on vient à parler de performances, il vient tout naturellement à l'esprit de penser à doter un langage fonctionnel de moyens en permettant l'exécution parallèle [Talpin & Jouvelot, 1993]. En FX, un parallélisme de données implicite peut être exprimé par l'utilisation d'un type de donnée abstrait: le vecteur, et d'un ensemble d'opérations globales associées, fortement inspirées de APL. Ici, l'inférence de type et d'effet s'avère être une méthode adéquate pour la mise en œuvre de ce langage de haut niveau, intégrant des paradigmes de programmation fonctionnelle et de programmation impérative, sur un calculateur massivement parallèle.

Dans l'implémentation d'un langage de programmation comme FX, qui intègre aussi bien des traits de programmation impérative que les concepts de programmation fonctionnelle, le concepteur doit apporter un soin tout particulier à la réalisation des techniques d'optimisation. En présence d'effets de bord, nombreuses sont en effet les propriétés formelles des programmes fonctionnels qui ne sont plus vérifiables, et nombreuses sont alors optimisations et transformations de programmes qui ne sont plus possibles ou deviennent non triviales.

L'analyse sémantique s'avère être un moyen efficace de remédier à cette carence. Mais cette technique devient d'une complexité significative dès lors qu'il s'agit de paralléliser des programmes, c'est à dire de

les optimiser de façon à produire un code exécutable sur des calculateurs massivement parallèles. En effet, la concomitance de constructions parallèles et impératives dans un programme conduit facilement au non-déterminisme. En pratique, le non-déterminisme est une propriété peu souhaitable dans la sémantique d'un langage de programmation [Steele, 1990]. Il rend plus difficile son utilisation en réduisant fortement la lisibilité des programmes. Il rend également la mise au point des programmes problématique; un résultat, et à fortiori une erreur, n'étant pas nécessairement reproductible. Au contraire, l'usage d'un parallélisme restreint à une forme déterministe dans un langage de programmation procure à son utilisateur une lecture aisée, car séquentielle, du texte des programmes, mais facilite surtout la spécification comme l'utilisation d'outils de développement comme de mise au point.

Nous étayons ces conjectures en présentant un ensemble de techniques de compilation mises en œuvre pour la compilation du langage FX sur un calculateur massivement parallèle, la CM-2. Notre compilateur utilise les effets afin de mettre en oeuvre correctement les opérations sur les vecteurs par un parallélisme de donnée: l'absence d'effets de bords, pour une opération distribuée sur les éléments d'un vecteur, garantit l'absence d'interférences lors de son exécution parallèle. Les autres opérations sont exécutées séquentiellement sur le frontal du calculateur. Notons qu'ici, et à l'instar d'autre compilateurs, cette optimisation est mise en oeuvre efficacement sans que la présence de traits impératifs ou de fonctions d'ordre supérieures ne pose de problèmes particuliers. Notre compilateur utilise les régions associées aux vecteurs afin de décider si la durée de vie des données que manipule un programme s'accommode avec l'organisation mémoire de la CM-2, dont l'architecture encourage fortement l'allocation de vecteurs parallèles en pile.

Structure de cette Thèse

L'étude développée dans cette thèse porte sur la spécification et la preuve d'une méthode d'analyse statique de programmes basée sur le principe de l'inférence de type, pour un langage de programmation inspiré de FX. Cette technique permet de déterminer les relations de partage entre les données manipulées par les programmes. Elle permet aussi de calculer les effets de bords provoqués par les constructions impératives du langage: celles qui modifient l'état d'exécution du programme.

Dans le chapitre 2, nous définissons tout d'abord le langage sur lequel notre étude sera basée, pour spécifier ensuite une méthode d'analyse des effets basée sur le principe de l'inférence de type. Nous définissons un algorithme permettant de calculer le type principal et l'effet minimal des expressions, affectant une région aux valeurs manipulées par le programme. Nous prouvons la conformité de cet algorithme à sa spécification, la sémantique statique.

Nous présentons ensuite une extension de ce système à d'autres traits impératifs, comme les canaux de communications. Nous spécifions l'extension de notre sémantique statique dont nous montrons la correction vis à vis de la sémantique dynamique propre à cette extension.

Dans le chapitre 3, nous définissons une discipline de typage polymorphe basée sur l'inférence d'effets qui prend en compte la localité des régions, introduisant une notion d'observabilité des effets.

Enfin, dans le chapitre 4, nous présentons l'inférence de type et d'effet comme un moyen correct et efficace de mettre en œuvre le langage FX, intégrant des paradigmes de programmation fonctionnelle et de programmation impérative, sur un calculateur massivement parallèle, la CM-2. Nous montrons comment utiliser le type et les effets des expressions d'un programme, afin d'en exploiter le parallélisme de données implicite, comme par exemple lors d'opérations portant sur des vecteurs.

Chapter 1

Introduction

In many scientific disciplines as well as in computer science, but perhaps more specifically in the area of programming languages, it is a widely established fact that theoretical research must be connected and validated with practical investigations and implementation techniques. To achieve this goal, it is very important to base the development of implementation techniques for programming languages on methods that allow them to be formalized simply and prove them correct. This care for guiding the development of programming tools using strictly formal techniques is not opposed to the effectiveness or to the ease of use of their design.

On the contrary, a strict formalization and a proof of correctness are the guarantees for the universality and the security of using a system consistently with its specification. This is thus profitable to the reliance of the user in programming tool and yields to a better productivity and quality of program development. Appropriate formal methods permit significant enrichment the expressive power of programming languages and systems by statically collecting a lot of information about user programs for their verification, documentation or optimization.

Static Typing

Among program analysis methods, static typing is undoubtly the most popular technique. Static typing detects the most common cause of execution errors in a program: the inconsistent use of a data structure (e.g. when the use of a datum is inconsistent with respect to the structure of that datum, like adding `1` to the boolean value `true`). Static type checking is achieved by approximating using a type the structure of every value manipulated in a program and by verifying that the use of every value conforms to its type.

The strength of static typing is that the successful type checking of a program guarantees the absence of type errors. Because typing rules must be simple and programs can be complex, type systems have an inherent weakness. They often reject programs that are otherwise correct, though they cannot be recognized as such. This tradeoff, between a suitable simplicity of the type system and its effectiveness, has motivated a lot of work. The introduction of polymorphism resulted in a notable progress in this direction. Polymorphism allows the static typing of generic functions.

Generic functions, such as the identity, can operate on data of different structures. They are interesting for a programmer because they can be reused in many programs without modification. For example, many list processing functions, such as `reverse`, for reversing the elements of a list, or `map`, for mapping a function on the elements of a list, are essentially generic. This holds for other data structures as well, such as hash tables, trees or graphs. Without the notion of polymorphism, static typing prohibits this reuse and necessitates duplication of functions as in Pascal, or the explicit definition and instantiation of generic functions as in Ada. Polymorphic typing supports the reuse of generic functions implicitly and without duplication of code.

Earlier work provides the basis for the automatic computation of types. In [Morris, 1968], the author showed that the successful static typing of a program consists of solving recursive linear equations related to the syntactic structure of the program. Then, [Hindley, 1969] developed a method for computing the type of expressions of the lambda-calculus by using a unification procedure [Robinson, 1965].

Finally, [Milner, 1978] introduced polymorphic type inference in the functional language ML, providing the first type discipline that permitted the polymorphic typing of functions and supported their automatic computation. Combining type polymorphism and automated type computation was a significant advance in programming language research and has been the subject of much theoretical investigation and practical developments.

Milner's type system expresses polymorphism in `let` syntactic constructs. Understanding the `let` as an abbreviation offers a simple explanation of polymorphism. Semantically, the expression `(let (x e) e'`) has the same meaning as `e'[x/e]`, the substitution of `e` for the free occurrences of `x` in `e'`. In typing the substituted expression `e'[x/e]`, each occurrence of the bound expression `e` may have a different type. In Milner's typoing discipline. this is expressed by a type scheme which is associated with `x` and represents the possible types of `e`.

Imperative Languages and Typing Disciplines

Polymorphic typing is appropriate for functional programming languages. But the imperative style of programming is usually defined as opposed to functional programming by the use of operations that, for example, permit in-place modification of mutable data structures, such as pointers. When adding imperative features to a language, it becomes necessary to introduce a notion of state to understand the meaning of programs. This suffices to invalidate our previous explanation of `let`-expressions as abbreviations: the expression `(let (x e) e'`) no longer has the same meaning as `e'[x/e]`.

To type pointers in ML, one can think of introducing the type $ref(\tau)$ to represent pointers referencing a value of type τ. Then, one can type the pointer initialization procedure `new` by $\tau \rightarrow ref(\tau)$, because it returns a pointer initialized to the given argument, which can have any type τ. However, once initialized, that pointer must always be associated with the same type. Otherwise, one could initialize it with an *int* value, then read it and claim it has type *bool*.

Just as references change the semantics of `let` expressions, they also necessitate a change in the way `let` expressions are typed. The types that must not be generalized are those that appear in the types of references allocated by `let`-bound expressions. Unfortunately, those types cannot be precisely determined. Thus, any static type system attempting to integrate the typing of references must use a conservative approximation of them.

Short of the ad-hoc techniques used in the first type inference systems, the imperative type discipline [Tofte, 1987, Tofte, 1990, Milner & al., 1990] is the classical way to deal with the problem of type generalization for polymorphic functional languages in the presence of non-referentially transparent constructs. Its extension, based on weak type variables [Appel & Mac Queen, 1990], is used in the implementation of Standard ML. A different approach [Leroy, 1990], consists of labeling the type of each function with the set of the types of the value identifiers that occur in its body, and then tracking the *dangerous* type variables on which side-effecting operations are performed.

All these approaches build conservative approximations of value types that may be accessible from the global store and turn out to be restrictive in practice by prohibiting generic functions that create temporary mutable structures or by being non-conservative over ML.

Effect Systems and Typing Disciplines

In the quest for integrating imperative constructs to polymorphic functional languages, inferring the type of stored values has been the subject of many investigations [Damas, 1985, O'Toole, 1990, Wright, 1992]. Effects systems, implemented by the FX language [Lucassen, 1987, Gifford & al., 1987], allow this integration. Sightly inspired by Scheme [Rees & al., 1988] and ML [Milner & al., 1990], the FX language supports higher-order functions, dynamic allocation and automatic deallocation of data, imperative constructs. FX also supports a system of first-class modules [Sheldon & Gifford, 1990].

But first of all, FX is an effect system. Just as types describe the structure of what expressions compute, effects describe how expressions compute. Types and effects are decorated with regions. A region describe an uniform sharing relation between data structures and thus helps to figure out how storage resources are

used and distributed in a program. With such an expressive power, the language FX allows programmers to express themselves very naturally in a purely *functional* style, as in ML, or in a more *imperative* style, as in more traditional programming languages, such as C or Fortran.

In [Gifford & al., 1987], the static semantics of FX is defined as the static checking of polymorphic type and effect declarations in FX programs. However, the need to specify types, regions and effects is burdensome in real-life programs. In [Jouvelot & Gifford, 1991], the first system for statically inferring effects is presented by introducing the notion of *algebraic reconstruction*.

1.1 Contribution

We introduce a new type system which, build upon both the ideas of effects systems [Gifford & al., 1987] and polymorphic type inference [Milner, 1978], and unlike algebraic reconstruction, reconstructs the principal type and the minimal effect of programs. As is explained in chapter 2, the addition of of a rule based on the relation of inclusion between effects to the static semantics, introduces a tantamount notion to subtyping in the domain of effects: *subeffecting*. Subeffecting enables to reconstruct the principal type and effect of expressions.

In this dissertation, type and effect inference is extended to the inference of regions. Regions appear in types. For example, $ref_\rho(\tau)$ is a pointer to a value of type τ in the region ρ. Regions appear in effects. $init(\rho)$ is the effect of initializing a pointer in the region ρ. Regions statically describe sharing relations between the values manipulated in a program. Incidentally, they permit to precisely delimit the scope-locality of data and allows to determine those which are subject to side-effects.

This observation has both practical and theoretical consequences that will be formally developed in this dissertation. On the practical side, a static information about the scope-locality of data permits to implement compile-time techniques for optimizing the management of their allocation and collection.

To demonstrate how effect systems allow the integration of imperative programming features in polymorphic functional languages, we introduce an imperative typing discipline that uses effect inference for determining the principal type of expressions: the type and effect discipline. The essential idea behind this new type system is to use effect inference for approximating the state transformation that is performed by side-effects, such as the allocation of a reference, a communication channel, a continuation.

Typing references is done by inferring allocation effects which tells the data type pointed at by regions of initialized references. Effects are used to control type generalization in the presence of imperative constructs. To type a `let` construct, the allocation effect of the bound expression provides all needed information to determine which type variables must not be generalized.

By using an observation criterion, our typing discipline limits the report of effects to those that affect accessible regions. The observable effects of an expression range over the regions that are free in its type environment and its type. Effects related to local data structures can be discarded during type reconstruction. The type of an expression can be generalized with respect to the variables that are not free in the type environment or in the observable effect.

The notion of observable effects is crucial to distinguishing the functions which only use references locally and implement their purely functional counterpart with a more imperative style. By using effect information together with an observation criterion, our type system is able to precisely delimit the scope of side-effecting operations, thus allowing type generalization to be performed in `let` expressions in a more efficient and uniform way than previous type systems.

1.2 Motivation

The academic interest in functional programming, such as ML [Milner & al., 1990], does not limit itself to the topics of polymorphic type inference. It is certainly due to the relationship between functional languages, the lambda calculus [Barendregt, 1984], and the ease of formalizing them using formal semantics methods [Plotkin, 1981, Stoy, 1977].

The strong relation between functional languages and lambda-calculi also encourages one to use proofs systems [Coquand & Huet, 1988, Huet, 1989, Harper & al., 1987], based on higher-order logics and lambda

calculi [Girard, 1972, Girard, 1986], for proving the correctness of compilation techniques [Hannan, 1990, Hannan & Pfenning, 1992] and program optimization [Wand, 1992] and for defining intermediate function representation [Wand, 1991].

Going from the dynamic semantics of functional language to their static semantics, polymorphic typing provides useful information for both the programmer, who can describe the intended specification of its programs, and the compiler, which can use types to generate more efficient code by avoiding type tags [Goldberg, 1991], to represent data structures unboxed [Leroy, 1990] or to help in garbage collection [Goldberg, 1992].

In a similar manner, effect inference permits uniform sharing relations between data structures and the side effects of programs to be determined. As is advocated in [Tofte & Talpin, 1993, Talpin & Jouvelot, 1993, Tang & Jouvelot, 1992], this information can be used to determine the allocation class of data structures: register, stack or heap. They can also be used to perform program transformations that can imply a change in the evaluation order of expressions in the program [Talpin & Jouvelot, 1993].

1.3 Implementation

To improve the performance of a programming language, one can also think of adding parallel execution capabilities to its implementation. For example in FX, implicit data parallelism can be expressed by using the vector module, consisting of an abstract data type and a set of functions for manipulating them, inspired by APL and [Fortran90].

Implementors of programming languages that integrates both functional and imperative paradigms must exert care when designing code optimizers since side effects inhibit most of the nice properties of pure functional languages, properties which are put at work in code transformations.

Going from sequential to parallel programs causes the issues to get significantly more complicated, both at the programming and the implementation levels. Concomitant use of side effects and parallelism leads to non-determinism, which makes program understanding and debugging difficult because of the non-reproductibility of results. Restricting parallel programs to be deterministic, as advocated in [Steele, 1990], is a way of making parallel program design in higher-order imperative languages a more manageable task.

Based on the concept of an effect system, we present new compile-time techniques that enforces such deterministic constraints and prove its effectiveness by describing a prototype compiler that targets the FX programming language [Gifford & al., 1987] to the Connection Machine CM-2.

Our compiler uses effects to determine when operations on vectors are amenable to data parallelism in the presence of both side effects and higher-order functions. The absence of side-effects, for an operation mapped on every element of a vector, guarantees that its execution in parallel will not cause interferences. Such operations are run in parallel while others are conservatorily limited to sequential execution on the CM-2 front end.

Our compiler uses regions to discover when the lifetime of locally allocated data structures is compatible with the memory model of the CM-2, which encourages the allocation of parallel vectors in the stack.

An implementation of these compile-time techniques has been integrated to the FX system, together with a CM-2 compiler back-end that generates *Lisp code. Test programs have been run on both a *Lisp simulator [*Lisp, 1987] and a CM-2 to evaluate the performance of our approach.

1.4 Structure of this Thesis

The contributions developed in this thesis aim at the specification and the proof of static analysis techniques, based on the principle of polymorphic type inference. The rest of the document is organized as follows.

In chapter 2, we present the language and its semantics. Then, we specify the static semantics for inferring types, regions, and effects and prove its consistency. We define the algorithm that computes the principal type and the minimal effect of expressions in the language. We prove its correctness with respect to the static semantics.

Then, we present some extensions of our system to capture other imperative features of programming under the concept of effect system, such as communications and concurrency. We specify suitable exten-

sions of our static semantics for incorporating them and prove their consistency with respect to a dynamic semantics.

In chapter 3, we show how to restrict us to the effects that affect their environment: the observable effects, and we present an application of effect inference, which leads to the definition of the *type and effect discipline*, an imperative typing discipline based on effect inference. This type discipline shows how to use effect information to control polymorphic type generalization in the presence of imperative language features.

Finally, in chapter 4, we present type and effect inference as a correct and effective medium for efficient implementation of the FX language on the CM-2. We show how to use type and effect information to exploit the implicit data-parallelism of FX vector operations and present a working compiler for FX on the CM-2.

Chapter 2

Type, Region and Effect Inference

2.1 Introduction

Static typing is the most widely used technique of static analysis in programming languages. It detects at compile-time a frequent cause of error during the execution of a program: the inconsistent use of a datum with respect to that datum's structure. The compiler uses types to statically describe the structure of data. Static typing associates every identifier and every expression of a program with a type according to certain rules. The process of type checking consists of verifying that the type of data, as they are used in the program, matches the type of their definition in the program. The successful type checking of a program guarantees that no type-error can ever occur during its execution.

However, an ill-typed program is not necessarily incorrect. This consideration has motivated the search for more expressive type checking systems, capable of rejecting less correct programs. The introduction of type polymorphism [Milner, 1978] has permitted such progress by allowing the typing of generic functions. Consequently, it has been the subject of much theoretical investigation and practical developments to integrate imperative language constructs [Damas, 1985, Tofte, 1987, Leroy, 1992, Talpin & Jouvelot, June 1992] and module systems [Appel & Mac Queen, 1990, Sheldon & Gifford, 1990, Tofte, 1992].

Effect systems [Lucassen & Gifford, 1988, Lucassen, 1987] are another example of such an extension. Similar to types, which describe what an expression computes, effects describe how an expression computes. Effect systems adapt type checking techniques to statically determine the type and side-effect of programs. In [Gifford & al., 1987], the authors define the FX programming language and propose a static semantics to check polymorphic type and effect declarations in FX programs. To spare user's from writing types, [Jouvelot & Gifford, 1991] presents a system for statically inferring types and effects. Introducing the notion of *algebraic reconstruction*, they formulate the problem of type and effect inference in terms of constraint satisfaction in the vein of [Morris, 1968].

The problem of constraint satisfaction is inherent to type inference. It consists of solving the relationships between types, specified by the static semantics, which are more often than not equations. The resolution of type equations is usually performed by using a unification procedure. When an equation has a solution, a unification procedure serves to compute the solutions of the equation. In the case of type equations, this solution is usually unique and represented by a substitution, which relates the variables of the equation to the terms that satisfy it.

Because unification is a central problem in type inference, the unicity of the solution to type equations is a very important formal property. But in general, it is far from being inherent to every unification problem. As is reviewed in [Siekmann, 1989], it essentially depends on the axiomatic properties of the algebra in which the equations are defined. ACUI-unification, studied by [Lincoln & Christian, 1989] and others, is a typical example of non-unitary unification problem. Some other equational problems, higher-order unification for instance, are even undecidable [Siekmann, 1989].

In an effect system, the side-effect of a function, its *latent effect*, is statically represented by a set associated with that function's type. As a consequence, the constraint satisfaction problem related to type and effect inference requires the resolution of equations on sets. Since the most general unifiers of a set

equation isn't, in general, reduced to a singleton, this implies that the principal type of an expression with respect to substitution is not always unique. Another problem of using algebraic reconstruction is that some programs, type-safe in Milner's type discipline, become ill-typed by the introduction of effect to the function types: effect mismatches cause type clashes.

In this chapter, our main contribution is to present a type system which building upon both the ideas of effects systems and polymorphic type inference, reconstructs the principal type and the minimal effect of such programs by the addition of subeffecting. *Subeffecting* is tantamount to subtyping in the domain of effects. It is required here to coerce the latent effect of matching functional types to a common upper bound. The algorithm presented in section 2.6 computes the minimum of these upper bounds.

Plan

The structure of this chapter is as follows. We first describe the static semantics of the language in section 2.3. In section 2.5, we prove that the static semantics of the language are consistent with the static semantics defined in 2.2. Section 2.6 presents our type, region and effect reconstruction algorithm. Its correctness is proved in section 2.8. In section 2.10, we present an extension of type and effect inference to capture other imperative features of programming under the concept of effect system, such as communications and concurrency. We specify suitable extensions of our static semantics for incorporating them and prove their consistency with respect to a dynamic semantics. Section 2.11 presents the related work. Before concluding in section 2.12, we show how our algorithm works on a few examples (section 2.9) and discuss potential applications. The technical results presented in this chapter are mainly inspired from [Talpin & Jouvelot, 1992].

2.2 A Core Language and its Semantics

Reasoning on the complete definition of the language FX or ML would have been complex and tedious. In order to simplify the presentation and to ease the formal reasoning, this chapter introduces a core language. It integrates, like ML or FX, the principal features of functional and imperative programming, but it is much simpler. This section introduces its syntax and its dynamic semantics together with a series of conventions and notations that are used in this thesis.

2.2.1 Syntax

The expressions of the language, written `e` possibly with a prime or a subscript, are the elements of the term algebra *Exp* generated by the grammar described below. It uses enclosing parentheses in the reminiscence of Scheme [Rees & al., 1988].

`e` ::=	`x` |	value identifier
	`(op e)` |	operation
	`(e e′)` |	application
	`(lambda (x) e)` |	abstraction
	`(rec (f x) e)` |	recursive function definition
	`(let (x e) e′)`	lexical value binding

Syntax

In this grammar, `x` and `f` range over a countable set of identifiers. The form `(e e′)` stands for the application of a function to an argument `e′`. The form `(op e)` applies the primitive operation `op` to the argument `e`. The expression `(lambda (x) e)` is the so-called lambda-abstraction that defines the first-class function whose parameter is `x` and whose result is the value of `e`. Similarly, the expression `(rec (f x) e)` defines a recursive function whose name is `f` inside `e`.

In the literature, the `lambda` construct is usually preferred to the `rec` construct, because it is easier to reason about functions with it and because it is simpler to handle in proofs. However, one can observe that it is hard to write interesting programs without recursion. Moreover, the addition of recursive functions to a language sometimes requires a complete revision of its specification and its proofs [Milner, 1991]. In the ML language, recursion is usually presented as an extension of the `let` construct. Here, the `let` construct is used to lexically bind a value identifier `x` to the value of an expression `e` during the evaluation of a body expression `e`.

2.2.2 Store operations

The arithmetic operations over integers: `+`, `-` and `<=`, the boolean operations: `and`, `or` and `not`, or even the construct `if` are typically represented by operators `op`, because their meaning cannot be explained by abstractions and applications.

Store operations can also be defined by operators. They operate on reference values, which are indirection cells that can be dynamically allocated, read and written in place.

`c` ::=	`unit`	value of commands
`op` ::=	`new` |	initialization
	`get` |	dereference
	`set`	assignment

Store Operations

The operation `(new e)` initializes a fresh reference to the value of the expression `e`. The operation `(get e)` gets the value referenced by the pointer returned by `e`. The operation `((set e) e'`) modifies the content of the reference returned by `e` and sets it to the value of `e'`. We use the convention that `set` returns the unit value u which is represented by the constant `unit` in the syntax of the language.

2.2.3 Free Value Identifiers

The set of free value identifiers $fi[\![\texttt{e}]\!]$ of an expression `e` is defined by induction on the syntax of expressions.

$$
\begin{aligned}
fi[\![\texttt{c}]\!] &= \emptyset \\
fi[\![\texttt{x}]\!] &= \texttt{x} \\
fi[\![\texttt{(op e)}]\!] &= fi[\![\texttt{e}]\!] \\
fi[\![\texttt{(e e')}]\!] &= fi[\![\texttt{e}]\!] \cup fi[\![\texttt{e'}]\!] \\
fi[\![\texttt{(lambda (x) e)}]\!] &= fi[\![\texttt{e}]\!] \setminus \{\texttt{x}\} \\
fi[\![\texttt{(rec (f x) e)}]\!] &= fi[\![\texttt{e}]\!] \setminus \{\texttt{f}, \texttt{x}\} \\
fi[\![\texttt{(let (x e) e')}]\!] &= fi[\![\texttt{e}]\!] \cup (fi[\![\texttt{e'}]\!] \setminus \{\texttt{x}\})
\end{aligned}
$$

Free Identifiers

2.2.4 Formulation of the Dynamic Semantics

In this section, we define the dynamic semantics of our language. The dynamic semantics specifies the meaning of the expressions of the language. It is defined by an evaluation mechanism that relates expressions to values. To express this relation, we use the formalism of relational semantics [Plotkin, 1981, Kahn, 1988]. It consists of a predicate between expressions and values defined by a set of axioms and inference rules: the evaluation judgement. This evaluation judgment tells whether an expression evaluates to a given result.

Another framework for expressing the meaning of a language's expressions is denotational semantics [Stoy, 1977], but it requires the formal definition of semantics domains for values which are not needed here. Reduction semantics [Felleisen & Friedman, 1989, Wright & Felleisen, 1992] is another simple approach to express the semantics of programming languages, via rewriting rules that use a notion of context.

2.2.5 Semantic Objects

We present the semantics objects on which the predicate of evaluation is defined. This semantic objects consist of values, environments, stores and traces.

v	$\in$	*Value*	$=\{u\}+Ref+Closure$	values
c	$\in$	*Closure*	$=Id\times Exp\times Env$	closures
l	$\in$	*Ref*		locations
E	$\in$	*Env*	$=Id \xrightarrow{fin} Value$	environments
s	$\in$	*Store*	$=Ref \xrightarrow{fin} Value$	stores
f	$\in$	*Trace*	$=\mathcal{P}(init(Ref)+read(Ref)+write(Ref))$	traces

Values, Stores and Traces

Values are either the command value u, reference values l or closures c. A closure $(\mathtt{x}, \mathtt{e}, E)$ is composed of a value identifier $\mathtt{x}$, its formal parameter, an expression $\mathtt{e}$, its body, and the environment E where it is defined.

An environment E is represented by a finite map from identifiers to values. In an environment E, we assume that all identifiers are distinct. The empty mapping is written $\{\}$. The domain of the mapping E is written $Dom(E)$ and its range $Im(E)$. If $\mathtt{x}$ belongs to $Dom(E)$, we write $E(\mathtt{x})$ for the value associated with $\mathtt{x}$ in E. Finally, we write $E_{\mathtt{x}}$ for the exclusion of $\mathtt{x}$ from E and the extension of E to the mapping of $\mathtt{x}$ to v by $E_{\mathtt{x}} + \{\mathtt{x} \mapsto v\}$.

The presence of references requires the introduction of a notion of state in the dynamic semantics: the store. The store changes during the evaluation of a program and it tells the current contents of all initialized references. We assume that we are given a countable set *Ref* of locations l. Then, a store s is represented by a finite map from references, or locations in *Ref* to values. Thus, we use for stores the same notations than for environments.

Additionally, we need to define traces in order to record side-effects that occur during evaluation. A trace f is represented by a set of tagged references $init(l)$, $read(l)$ and $write(l)$ that indicate initialized, read and written locations. A trace is intended to describe the dynamic counterpart of a static effect presented in section 2.3.

2.2.6 Formulation of Axioms and Rules

The definition of the dynamic semantics is presented by a set of axioms and inference rules. Axioms and rules are made of propositions P. Propositions P in axioms and rules can contain variables that are implicitly universally quantified. An axiom, presented as:

$$\frac{}{P} \qquad \text{or sometimes as} \qquad P$$

allows to conclude that proposition P holds for any instance of its free variables, i.e. for any substitution of its free variables with appropriate ground terms. Similarly, an inference rule:

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{P}$$

allows one to conclude that P holds when the premises $P_1 \ldots P_n$ have been proved.

2.2.7 Axioms and Rules of the Dynamic Semantics

We present, in the vein of [Tofte, 1987, Milner, 1991], the set of rules that inductively defines the predicate of evaluation $s, E \vdash \mathtt{e} \rightarrow v, f, s'$ on the structure of expressions. Given a store s and an environment E, the predicate $s, E \vdash \mathtt{e} \rightarrow v, f, s'$ associates each expression $\mathtt{e}$ with a value v, a trace of side-effects f and a new store s'.

$$s, E \vdash \mathtt{x} \rightarrow E(\mathtt{x}), \emptyset, s \qquad \text{(var)}$$

$$s, E \vdash \texttt{(lambda (x) e)} \rightarrow (\mathtt{x}, \mathtt{e}, E_{\mathtt{x}}), \emptyset, s \qquad \text{(abs)}$$

$$\frac{c = (\mathtt{x}, \mathtt{e}, E_{\mathtt{f},\mathtt{x}} + \{\mathtt{f} \mapsto c\})}{s, E \vdash \texttt{(rec (f x) e)} \rightarrow c, \emptyset, s} \qquad \text{(rec)}$$

$$\frac{s, E \vdash \mathtt{e} \rightarrow v, f, s' \qquad s', E_{\mathtt{x}} + \{\mathtt{x} \mapsto v\} \vdash \mathtt{e}' \rightarrow v', f', s''}{s, E \vdash \texttt{(let (x e) e')} \rightarrow v', f \cup f', s''} \qquad \text{(let)}$$

$$\frac{s, E \vdash \mathtt{e} \rightarrow (\mathtt{x}, \mathtt{e}'', E'), f, s' \qquad s', E \vdash \mathtt{e}' \rightarrow v', f', s'' \qquad s'', E' + \{\mathtt{x} \mapsto v'\} \vdash \mathtt{e}'' \rightarrow v'', f'', s'''}{s, E \vdash \texttt{(e e')} \rightarrow v'', f \cup f' \cup f'', s'''} \qquad \text{(app)}$$

Dynamic Semantics

The axiom (var) states that an identifier $\mathtt{x}$ evaluates to the value $E(\mathtt{x})$ bound to it in the environment E, provided that this identifier $\mathtt{x}$ belongs to the domain of E. Otherwise, the expression $\mathtt{x}$ has no meaning. By the axiom (abs), a function definition evaluates to a closure.

The rule (rec) is more involved. A similar representation can be found in [Milner, 1991]. The equation $c = (\mathtt{x}, \mathtt{e}, E_{\mathtt{f},\mathtt{x}} + \{\mathtt{f} \mapsto c\})$, that defines the closure c representing the function $\mathtt{f}$, is recursive. To support this feature, we need to define non well-founded semantics objects c in *Closure*. Following [Aczel, 1988], it is possible to define the domains *Closure*, *Value* and *Env* that satisfy the recursive set equations that define them and, also, in such a way that there exists a unique c satisfying the equation $c = (\mathtt{x}, \mathtt{e}, E_{\mathtt{f},\mathtt{x}} + \{\mathtt{f} \mapsto c\})$ for any $\mathtt{x}$, $\mathtt{f}$, $\mathtt{e}$ and E.

As stated by the rule (let), a let binding evaluates the first argument $\mathtt{e}$ to a value v, binds it to the identifier $\mathtt{x}$, and then evaluates its second argument $\mathtt{e}'$ in the environment E extended with $\{\mathtt{x} \mapsto v\}$. The result v' is the result of the let expression. In the case that the evaluation of the first argument does not succeed, the evaluation of the let expression is not defined.

The rule of application (app) is more complex. First, the expression $\mathtt{e}$ must evaluate to a closure $(\mathtt{x}, \mathtt{e}'', E')$. Then, the argument $\mathtt{e}'$ must evaluate to a value v'. Finally, the function body $\mathtt{e}''$ must evaluate to a value v'' with the environment E', captured in the closure, extended with the formal parameter $\mathtt{x}$ bound to v'. In the case of a recursive function, defined with the rule (rec), the internal function name $\mathtt{f}$ is still bound to the closure $(\mathtt{x}, \mathtt{e}'', E')$, thus allowing recursive calls.

In the rules (let) and (app), the traces of the side effects f, f' and f'', occurring in during the evaluation of subexpressions $\mathtt{e}$, $\mathtt{e}'$ and $\mathtt{e}''$, are collected and combined together by the set union operator $\cup$.

2.2.8 Dynamic Semantics of Store Operations

Now, we can give the relational semantics for the operations on references. The semantics describe how the store is modified by the evaluation of expressions. Additionally, it gives a precise definition of side-effects which are collected by evaluation traces f and consist of initializations $init(l)$, dereferences $read(l)$ or assignments $write(l)$ of locations l.

$$\frac{s, E \vdash \texttt{e} \rightarrow v, f, s' \qquad l \notin Dom(s')}{s, E \vdash \texttt{(new e)} \rightarrow l, f \cup \{init(l)\}, s' + \{l \mapsto v\}} \quad \text{(new)}$$

$$\frac{s, E \vdash \texttt{e} \rightarrow l, f, s' \qquad l \in Dom(s')}{s, E \vdash \texttt{(get e)} \rightarrow s'(l), f \cup \{read(l)\}, s'} \quad \text{(get)}$$

$$\frac{s, E \vdash \texttt{e} \rightarrow l, f, s' \qquad s', E \vdash \texttt{e}' \rightarrow v, f', s''}{s, E \vdash \texttt{((set e) e')} \rightarrow u, f \cup f' \cup \{write(l)\}, s''_l + \{l \mapsto v\}} \quad \text{(set)}$$

Dynamic Semantics of Store Operations

The rule (new) of reference initialization first evaluates the initial value v of the reference and then picks a fresh location l. This very step might seem non-deterministic but all choices of l are equivalent modulo a renaming of the locations in s'. The second step is then to extend the store with the binding of l to v and to return l as the value of the expression. The rule (get) evaluates its argument `e` to a location l, then returns the value v stored at this location in the store s. Finally, by the rule (set), the assignment operator evaluates its first argument `e` to a location l and its second argument `e'` to a value v. Then, it updates the store at the location l, substituting the previous value by v. Note that, by definition of the rule (new), l must be in s when `e` evaluates to l.

$$\frac{s, E \vdash \texttt{e} \rightarrow l, f, s'}{s, E \vdash \texttt{(set e)} \rightarrow (set, l), f, s'} \quad (\text{set}_1)$$

$$\frac{s, E \vdash \texttt{e} \rightarrow (set, l), f, s' \qquad s', E \vdash \texttt{e}' \rightarrow v, f', s''}{s, E \vdash \texttt{(e e')} \rightarrow u, f \cup f' \cup \{write(l)\}, s''_l + \{l \mapsto v\}} \quad (\text{set}_2)$$

Another Dynamic Semantics of Assignment

The particular syntax of the assignment operator in the rule (set) is here to avoid complicating the static semantics, presented in section 2.3, by introducing functions with multiple arguments. Alternatively, we could present it by considering `new`, `get` and `set` as identifiers as well as introducing the partial applications (set, l) of the `set` operator to locations l.

Example Before moving to the technical developments of this thesis, let us demonstrate how to use the dynamic semantics by considering the derivation of the small program below.

$$\{\}, \{\} \vdash \texttt{(lambda (x) (get (new x)))} \rightarrow (\texttt{x}, \texttt{(get (new x))}, \{\}), \{\}, \{\}$$

$$\{\}, \{\} \vdash \texttt{1} \rightarrow 1, \{\}, \{\}$$

$$\dfrac{\dfrac{\{\}, \{\texttt{x} \mapsto 1\} \vdash \texttt{x} \rightarrow 1, \{\}, \{\} \qquad l \text{ is fresh}}{\{\}, \{\texttt{x} \mapsto 1\} \vdash \texttt{(new x)} \rightarrow l, \{init(l)\}, \{l \mapsto 1\}}}{\{\}, \{\texttt{x} \mapsto 1\} \vdash \texttt{(get (new x))} \rightarrow 1, \{init(l), read(l)\}, \{l \mapsto 1\}}$$

$$\frac{}{\{\}, \{\} \vdash \texttt{((lambda (x) (get (new x))) 1)} \rightarrow 1, \{init(l), read(l)\}, \{l \mapsto 1\}}$$

Our program executes in the empty store and the empty environment (both written $\{\}$). It is an application expression. The left-hand side evaluates to a closure $(\texttt{x}, \texttt{(get (new x))}, \{\})$ and the right-hand side to an integer constant, 1. Then, the closure $(\texttt{x}, \texttt{e}, \{\})$ is applied to its argument 1 and performs two pointer operations. First, it allocates a fresh pointer l and sets it to 1 by the operation `new`. It then reads and returns it. The program terminates, yielding the value 1. Also note that the location l, in which the value 1 was stored, cannot be accessed any longer ■

2.3 Static Semantics

In this section, we present the static semantics of our language. We are first going to equip the language with a type system. Then we will give the inference rules of the static semantics. The rules of the static semantics associate the expressions of the language with their type and effect, in the same way as the rules of the dynamic semantics associate expressions with values.

2.3.1 Semantic Objects

We begin by defining the term algebra for the three basic kinds of semantic objects: regions, effects and types.

$$
\begin{array}{lll}
\rho & ::= r \mid \varrho & \text{regions} \\
\sigma & ::= \emptyset \mid \varsigma \mid \sigma \cup \sigma \mid \mathit{init}(\rho) \mid \mathit{read}(\rho) \mid \mathit{write}(\rho) & \text{effects} \\
\tau & ::= \mathit{unit} \mid \alpha \mid \mathit{ref}_{\rho}(\tau) \mid \tau \xrightarrow{\sigma} \tau & \text{types}
\end{array}
$$

Static Semantics Objects

The domain *Region* of regions ρ is the disjoint union of a countable set of constants r and variables ϱ. Every location corresponds to a given region in the static semantics. A region abstracts the memory locations that will be initialized at a given program point at runtime.

Basic effects σ can either be the constant $\emptyset$ that represents the absence of effects, effect variables ς, or store effects $\mathit{init}(\rho)$, $\mathit{read}(\rho)$ or $\mathit{write}(\rho)$ that approximate memory side-effects on their region argument ρ. $\mathit{init}(\rho)$ denotes the allocation and initialization of a mutable reference value in the region ρ. The effect $\mathit{read}(\rho)$ describes accesses to references in the region ρ, while $\mathit{write}(\rho)$ represents assignments of values to references in the region ρ.

Effects can be gathered together with the infix operator $\cup$ that denotes the union of effects; effects define a set algebra. The equality on effects is thus defined modulo associativity, commutativity and idempotence with $\emptyset$ as the neutral element. We define the set-inclusive relation $\supseteq$ of subsumption on effects: $\sigma \supseteq \sigma'$ if and only if there exists an effect σ'' such that $\sigma = \sigma' \cup \sigma''$.

The domain *Type* of types τ is composed of the constant *unit*, which describes the type of commands, type variables α, reference types $\mathit{ref}_{\rho}(\tau)$ in region ρ to values of type τ, function types $\tau \xrightarrow{\sigma} \tau'$ from τ to τ' with a *latent effect* σ. The latent effect of a function encapsulates the side-effects of its body and is the effect incurred when the function is applied.

2.3.2 Free Variables and Substitutions

We have defined three kinds of variables: type variables, region variables and effect variables. When it is not necessary to specify if a variable represents a type, a region or an effect, we note it v generically. Also, we adopt to represent sequences of terms, such as sequences of variables v, using the notation $\vec{v}$.

We write $\mathit{fv}(\tau)$ for the set of free type, region and effect variables in τ. This definition extends pointwise to regions and effects.

$$
\begin{array}{rlrl}
\mathit{fv}(\mathit{unit}) &= \emptyset & \mathit{fv}(\emptyset) &= \emptyset \\
\mathit{fv}(\alpha) &= \{\alpha\} & \mathit{fv}(\mathit{init}(\rho)) &= \mathit{fv}(\rho) \\
\mathit{fv}(\mathit{ref}_{\rho}(\tau)) &= \mathit{fv}(\rho) \cup \mathit{fv}(\tau) & \mathit{fv}(\mathit{read}(\rho)) &= \mathit{fv}(\rho) \\
\mathit{fv}(\tau \xrightarrow{\sigma} \tau') &= \mathit{fv}(\tau) \cup \mathit{fv}(\tau') \cup \mathit{fv}(\sigma) & \mathit{fv}(\mathit{write}(\rho)) &= \mathit{fv}(\rho) \\
\mathit{fv}(r) &= \emptyset \mid \mathit{fv}(\varrho) = \{\varrho\} & \mathit{fv}(\sigma \cup \sigma') &= \mathit{fv}(\sigma) \cup \mathit{fv}(\sigma')
\end{array}
$$

Free Variables

The function fr computes the set of region constants and variables free in type and effect terms. Similarly, we write ftv, frv and fev for free type variables, region variables and effect variables of terms respectively.

$$
\begin{array}{rlrl}
fr(\mathit{unit}) &= \emptyset & fr(\emptyset) &= \emptyset \\
fr(\alpha) &= \{\alpha\} & fr(\mathit{init}(\rho)) &= \{\rho\} \\
fr(\mathit{ref}_{\rho}(\tau)) &= \{\rho\} \cup fr(\tau) & fr(\mathit{read}(\rho)) &= \{\rho\} \\
fr(\tau \xrightarrow{\sigma} \tau') &= fr(\tau) \cup fr(\tau') \cup fr(\sigma) & fr(\mathit{write}(\rho)) &= \{\rho\} \\
fr(\rho) &= \{\rho\} & fr(\sigma \cup \sigma') &= fr(\sigma) \cup fr(\sigma')
\end{array}
$$

Free Regions

Substitutions θ map type variables α to types τ, region variables ϱ to regions ρ and effect variables ς to effects σ. We write $\theta \circ \theta'$ for the composition of the substitution θ and θ', so that $\theta \circ \theta'(\upsilon) = \theta(\theta'(\upsilon))$. The identity is written Id.

2.3.3 Rules of the Static Semantics

We formulate type and effect inference by a deductive proof system that assigns a type and an effect to every expression of the language. The context in which an expressions is associated with a type and an effect is represented by a type environment $\mathcal{E}$ which maps value identifiers to types. Deductions produce conclusions of the form $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ which are called typing judgment and reads "in the type environment $\mathcal{E}$ the expression `e` has type τ and effect σ".

$$\frac{\mathtt{x} \in \mathit{Dom}(\mathcal{E})}{\mathcal{E} \vdash \mathtt{x} : \mathcal{E}(\mathtt{x}), \emptyset} \quad \text{(var)}$$

$$\frac{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma}{\mathcal{E} \vdash \texttt{(lambda (x) e)} : \tau \xrightarrow{\sigma} \tau', \emptyset} \quad \text{(abs)}$$

$$\frac{\mathcal{E}_{\mathtt{f}} + \{\mathtt{f} \mapsto \tau\} \vdash \texttt{(lambda (x) e)} : \tau, \emptyset}{\mathcal{E} \vdash \texttt{(rec (f x) e)} : \tau, \emptyset} \quad \text{(rec)}$$

$$\frac{\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\sigma''} \tau', \sigma \qquad \mathcal{E} \vdash \mathtt{e}' : \tau, \sigma'}{\mathcal{E} \vdash \texttt{(e e')} : \tau', \sigma \cup \sigma' \cup \sigma''} \quad \text{(app)}$$

$$\frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \qquad \sigma' \supseteq \sigma}{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'} \quad \text{(does)}$$

Static Semantics

The typing rule (var), for value identifiers, states that an identifier `x` has a type τ as soon as it is bound to it in the type environment $\mathcal{E}$. The rule (abs), for abstraction, tells that a function definition has type $\tau \xrightarrow{\sigma} \tau'$ as soon as its body can be given type τ' and effect σ under the assumption its formal parameter is of type τ. In the case of (rec), the internal value identifier of the function must also have that type. Note that non-expansive expressions, such as value identifiers, rule (var) and abstractions, rules (abs) and (rec), have no effects.

In the rule (app), we see that a function application is well-typed as soon as the type of the argument corresponds to the type of the function parameter. The type of the application is the type given for the result of the function. The effect of the function is propagated together with the effects of both subexpressions.

The rule (does) is precisely needed because of the rule (app). The rules (app) imposes that the types of the formal parameter and of its argument match. Type-matching may occasion effect-matching which can

be avoided by a tantamount notion to subtyping in the domain of effects: *subeffecting*. Subeffecting, via the (does) rule, allows the extension of effects.

The communication of the effects from a function definition to a function application is best viewed in the rules of abstraction (abs) and application (app), which show the interesting interplay between types and effects. Via the abstraction rule, the effect of a lambda abstraction body is put inside the function type while, with the application rule, this embedded effect is extracted from the function type to be exercised at the point of call; effects flow from the points where functions are defined to the points where they are used.

2.3.4 Static Semantics of Store Operations

The store operations `new`, `get` and `set` have been defined by appropriate rules in the dynamic semantics. In the static semantics, they are best defined by using axioms.

$$\mathcal{E} \vdash \texttt{new} : \tau \xrightarrow{\sigma \cup init(\rho)} ref_\rho(\tau), \emptyset$$
$$\mathcal{E} \vdash \texttt{get} : ref_\rho(\tau) \xrightarrow{\sigma \cup read(\rho)} \tau, \emptyset$$
$$\mathcal{E} \vdash \texttt{set} : ref_\rho(\tau) \xrightarrow{\sigma} \tau \xrightarrow{\sigma' \cup write(\rho)} unit, \emptyset$$

Static Semantics for Imperative Operations

These three axioms specify the types that can be assigned to the identifiers `new`, `get` and `set` that implement store operations. For instance, the axiom for the identifier `new` reads: for any environment $\mathcal{E}$, type τ, region ρ and effect σ, the type of `new` is a function from objects of type τ to references $ref_\rho(\tau)$ that has at least the effect $init(\rho)$ of initializing a reference in the region ρ.

2.3.5 Static Semantics of let-binding Expressions

The notion of type polymorphism is the most distinguished feature of Milner's discipline of typing, introduced in [Milner, 1978]. It is expressed in `let` constructs. It reflects the property that an expression can have several different types. Generic types are associated with the value identifiers that are bound to referentially transparent expressions in `let` constructs.

One way to statically enforce such expressions be "referentially transparent" would be to require their effects to be $\emptyset$. We did not adopt this policy here since it would have required a non-deterministic backtrack-based inference algorithm. This would have departed too much from existing syntax-directed type reconstruction algorithms. Nonetheless, as shown in [Wright, 1992] and [Talpin & Jouvelot, June 1992], effects can be used to control the generalization of types.

The treatment of type polymorphism by effect inference is the subject of chapter 4. The present chapter introduces a much simpler policy for the introduction of type polymorphism, based on the notion of *expansiveness* of expressions [Tofte, 1987]. Informally, value identifiers and lambda-abstractions are non-expansive expressions. By extension, a `let` expression is non-expansive if and only if both its binding expression and its body are non-expansive.

$$\begin{aligned} exp[\![\texttt{e}]\!] = \quad & \text{case } \texttt{e} \text{ of} \\ & \texttt{x} \mid \texttt{(lambda (x) e)} \mid \texttt{(rec (f x) e)} \Rightarrow \text{false} \\ & \texttt{(e e')} \mid \texttt{(op e)} \Rightarrow \text{true} \\ & \texttt{(let (x e) e')} \Rightarrow exp[\![\texttt{e}]\!] \vee exp[\![\texttt{e'}]\!] \end{aligned}$$

Expansive Expressions

Non-expansive `(let (x e) e′)` expressions can be handled by the syntactic substitution `e′[e/x]` of `e` for `x` in `e′`, avoiding capture of bound variables. This simple technique provides an equivalent way of expressing the property that non-expansive expressions may admit multiple types without adding the complications of introducing type schemes in the static semantics.

$$\frac{\neg exp[\![\mathtt{e}]\!] \qquad \mathcal{E} \vdash \mathtt{e} : \tau, \emptyset \qquad \mathcal{E} \vdash \mathtt{e'[e/x]} : \tau', \sigma'}{\mathcal{E} \vdash \mathtt{(let\ (x\ e)\ e')} : \tau', \sigma'} \quad \text{(let)}$$

$$\frac{exp[\![\mathtt{e}]\!] \qquad \mathcal{E} \vdash \mathtt{((lambda\ (x)\ e')\ e)} : \tau, \sigma}{\mathcal{E} \vdash \mathtt{(let\ (x\ e)\ e')} : \tau, \sigma} \quad \text{(ilet)}$$

Static Semantics of `let` Expressions

By the rules (let) and (ilet) above, the typing of `let` expressions reduces to simpler expression by using the simple rewriting rules of syntactic substitution `e′[e/x]` and a syntactic expansiveness criterion.

Example We have now completely presented the dynamic and static semantics of our language. To exercise us at manipulating type and effect of expressions, let us consider the derivation of the type and the effect of the following small program, that computes the factorial of 5.

```
(let (x (new 1))
  ((rec (loop y) (if (= y 1) (get x)
                             (begin ((set x) (* y (get x)))
                                    (loop (- y 1)))))
   5))
```

In this example, we introduce two additional language constructs. The form `(begin e e′)` performs the sequential evaluation of `e` and `e′`. The form `(if e e′ e″)` chooses to evaluate either `e′` or `e″` depending on the boolean value of `e`.

Our program consist of an expansive `let` construct. Typing the bound expression `(new 1)` is no problem, it is an application of a store operation to an integer constant.

$$\frac{\{\} \vdash \mathtt{new} : int \xrightarrow{init(\rho)} ref_\rho(int), \emptyset \qquad \{\} \vdash \mathtt{1} : int, \emptyset}{\{\} \vdash \mathtt{(new\ 1)} : ref_\rho(int), init(\rho)}$$

To type the identifier `new`, we choose the type int in order to match the type of the constant 1 and verify the rule (app). There is no constraint in the choice of the region ρ and the effect $init(\rho)$ of `new`.

Having typed the bound expression, we must then, according to the rule (ilet), build a type environment $\mathcal{E} = \{\mathtt{x} \mapsto ref_\rho(int)\}$ and type the body of the expression. This body expression is an application which consists of a recursive function and another integer constant. Typing this constant is easy, we have $\mathcal{E} \vdash \mathtt{5} : int, \emptyset$. This is not the case for the recursive function.

According to the rule (rec), an hypothesis must be made about the type of the recursive function. The definition of the function must then fulfill this hypothesis.

$$\mathcal{E}' = \mathcal{E} + \{\mathtt{loop} \mapsto int \xrightarrow{read(\rho) \cup write(\rho)} int, \mathtt{y} \mapsto int\}$$

We choose the type of the function `loop` to be $int \xrightarrow{read(\rho) \cup write(\rho)} int$ and build the new type environment $\mathcal{E}'$ that will be used during the derivation of the type and the effect of the function definition.

$$\frac{\dfrac{}{\mathcal{E}' \vdash \mathtt{=} : int \times int \xrightarrow{\emptyset} bool, \emptyset} \quad \dfrac{\mathtt{y} \in Dom(\mathcal{E}')}{\mathcal{E}' \vdash \mathtt{y} : \mathcal{E}'(\mathtt{y}), \emptyset} \quad \dfrac{}{\mathcal{E}' \vdash \mathtt{1} : int, \emptyset}}{\mathcal{E}' \vdash \mathtt{(=\ y\ 1)} : bool, \emptyset}$$

We first type the predicate expression in the `if` construct, which is quite simple to do. The identifier `=` stands here for an equality predicate between integer values. In the static semantics, it can be associated with an axiom in the same vein than store operations.

$$\frac{\mathcal{E}' \vdash \texttt{get} : ref_\rho(int) \stackrel{read(\rho)}{\rightarrow} int, \emptyset \qquad \mathcal{E}' \vdash \texttt{x} : \mathcal{E}'(\texttt{x}), \emptyset}{\mathcal{E}' \vdash \texttt{(get x)} : int, read(\rho)}$$

Then, we type the left arm of the `if` construct and finally, the right arm, which is decomposed in a sequence of two operations. The first is an accumulation of the intermediate result using the pointer `x`.

$$\frac{\mathcal{E}' \vdash \texttt{(set x)} : int \stackrel{write(\rho)}{\rightarrow} unit, \emptyset \qquad \mathcal{E}' \vdash \texttt{(* y (get x))} : int, read(\rho)}{\mathcal{E}' \vdash \texttt{((set x) (* y (get x)))} : unit, read(\rho) \cup write(\rho)}$$

The second is the recursive call of the function `loop`.

$$\frac{\mathcal{E}' \vdash \texttt{loop} : \mathcal{E}'(\texttt{loop}), \emptyset \qquad \mathcal{E}' \vdash \texttt{(- y 1)} : int, \emptyset}{\mathcal{E}' \vdash \texttt{(loop (- y 1))} : int, read(\rho) \cup write(\rho)}$$

This gives the type of the `begin` construct and, finally, of the `if` construct.

$$\frac{\mathcal{E}' \vdash \texttt{(= y 1)} : bool, \emptyset \quad \mathcal{E}' \vdash \texttt{(get x)} : int, read(\rho) \quad \mathcal{E}' \vdash \texttt{(begin ...)} : int, read(\rho) \cup write(\rho)}{\mathcal{E}' \vdash \texttt{(if (= y 1) (get x) (begin ...))} : int, read(\rho) \cup write(\rho)}$$

The type of the definition of `loop` agrees with the type that was assigned to the identifier `loop`. Thus, the rule (rec) is verified. In the rest of our program, the recursive function `loop` is applied to its argument, 5, and thus yields a result of type *int* with the effects of initializing, reading and writing a pointer in the region ρ ■

2.4 Formal Properties of the Static Semantics

The static semantics presented in the previous section has a number of formal properties that are used in the formal proofs of consistency, with respect to the dynamic semantics (section 2.2), and of correctness, with respect to algorithm (section 2.6).

2.4.1 Substitution Lemma

An important formal property of the static semantics is the lemma of substitution. It is used both in the proof of consistency and in the proofs of correctness for the reconstruction algorithm.

Lemma 2.1 (Substitution) *If* $\mathcal{E} \vdash \texttt{e} : \tau, \sigma$ *then* $\theta\mathcal{E} \vdash \texttt{e} : \theta\tau, \theta\sigma$ *for any substitution* θ

Proof The proof is by induction on the proof derivation.

Case of (var) By hypothesis, we have $\mathcal{E} \vdash \texttt{x} : \tau, \emptyset$. By definition of the rule (var), $\tau = \mathcal{E}(\texttt{x})$. Thus, $\theta\tau = \theta(\mathcal{E}(\texttt{x}))$ and by definition of the rule (var), $\theta\mathcal{E} \vdash \texttt{x} : \theta\tau, \emptyset$.

Case of (abs) By hypothesis, we have $\mathcal{E} \vdash \texttt{(lambda (x) e)} : \tau \stackrel{\sigma}{\rightarrow} \tau', \emptyset$. By definition of the rule (abs), $\mathcal{E}_\texttt{x} + \{\texttt{x} \mapsto \tau\} \vdash \texttt{e} : \tau', \sigma$. By induction hypothesis on `e`, $\theta(\mathcal{E}_\texttt{x} + \{\texttt{x} \mapsto \tau\}) \vdash \texttt{e} : \theta\tau', \theta\sigma$ for any substitution θ. By definition of the rule (abs), $\theta\mathcal{E} \vdash \texttt{(lambda (x) e)} : \theta(\tau \stackrel{\sigma}{\rightarrow} \tau'), \emptyset$.

Case of (rec) By hypothesis, we have $\mathcal{E} \vdash \texttt{(rec (f x) e)} : \tau, \emptyset$. By definition of the rule (rec), $\mathcal{E}_\texttt{f} + \{\texttt{f} \mapsto \tau\} \vdash \texttt{(lambda (x) e)} : \tau, \emptyset$. By induction hypothesis on `(lambda (x) e)`, $\theta(\mathcal{E}_\texttt{f} + \{\texttt{f} \mapsto \tau\}) \vdash$ `(lambda (x) e)` $: \theta\tau, \emptyset$ for any θ. By definition of the rule (rec), $\theta\mathcal{E} \vdash \texttt{(rec (f x) e)} : \theta\tau, \emptyset$.

Case of (app) By hypothesis, $\mathcal{E} \vdash$ `(e e')` $: \tau', \sigma \cup \sigma' \cup \sigma''$. By definition of the rule (app), $\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\sigma''} \tau', \sigma$ and $\mathcal{E} \vdash \mathtt{e}' : \tau, \sigma'$. By induction hypothesis on e and e′, $\theta\mathcal{E} \vdash \mathtt{e} : \theta(\tau \xrightarrow{\sigma''} \tau'), \theta\sigma$ and $\theta\mathcal{E} \vdash \mathtt{e}' : \theta\tau, \theta\sigma'$ for any substitution θ. By the definition of the rule (app), we conclude that $\theta\mathcal{E} \vdash$ `(e e')` $: \theta\tau', \theta(\sigma \cup \sigma' \cup \sigma'')$ for any substitution θ.

Case of (does) By hypothesis, $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$. By definition of the rule (does), this requires that there exists $\sigma' \subseteq \sigma$ such that $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'$. By induction hypothesis, $\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \theta\sigma'$ for any substitution θ. By definition of the rule (does), we conclude that $\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \theta\sigma$ since $\theta\sigma' \subseteq \theta\sigma$ □

2.4.2 Deterministic Deduction

The typing rules presented here are derived from Damas & Milner's type discipline for ML. However, the system of [Damas & Milner, 1982] assigns type schemes to expressions and features two separate rules for generalization and instantiation. The use of these two rules is not constrained by the syntax, whereas in [Clément & al., 1985, Tofte, 1987], instantiation is performed for value identifiers and generalization is performed in `let` expressions. Every rule is associated with a syntactic category.

This is not the case in our system, because the rule (does), can be used any time during the deduction, and thus introduces non-determinism in the assignment of type and effect. This rule is necessary to coerce, to a common upper bound, the latent effect of two functions that must have matching types. However, by combining the rule (does) with the rule (abs) as above, we obtain a syntax-directed deduction system:

$$\frac{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash_{\mathtt{d}} \mathtt{e} : \tau', \sigma}{\mathcal{E} \vdash_{\mathtt{d}} \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\sigma \cup \sigma'} \tau', \emptyset} \quad \text{(does)+(abs)}$$

Proposition 2.1 (Deterministic deduction) *If* $\mathcal{E} \vdash_d \mathtt{e} : \tau, \sigma$ *then* $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$. *If* $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ *then* $\mathcal{E} \vdash_d \mathtt{e} : \tau, \sigma'$ *for some* $\sigma' \subseteq \sigma$.

Proof We show by induction that every derivation in $\vdash_d$ corresponds to a derivation in $\vdash$. Derivations using the rules (var), (app), (let) and (ilet) are equivalent. Derivations using the rule (abs) (respectively (rec)) can be translated by using the rules (does) and (abs) (respectively (rec)) as in the following case analysis.

Case of (abs) By hypothesis $\mathcal{E} \vdash_{\mathtt{d}}$ `(lambda (x) e)` $: \tau \xrightarrow{\sigma \cup \sigma'} \tau', \emptyset$. By the rule (abs), this requires that $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash_{\mathtt{d}} \mathtt{e} : \tau', \sigma$. By induction on e, we have $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma$. By the rule (does), we get $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma \cup \sigma'$. By the rule (abs), we conclude

$$\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\sigma \cup \sigma'} \tau', \emptyset$$

Now suppose that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau \xrightarrow{\sigma \cup \sigma'} \tau', \emptyset$. By the rule (abs), this requires that $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma$. By induction on e, we have $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash_{\mathtt{d}} \mathtt{e} : \tau', \sigma'$ for some $\sigma' \subseteq \sigma$. By the rule (does)+(abs), we conclude

$$\mathcal{E} \vdash_{\mathtt{d}} \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\sigma \cup \sigma'} \tau', \emptyset \qquad \square$$

2.5 Consistency of Dynamic and Static Semantics

We use the proof techniques introduced in [Tofte, 1987, Milner, 1991] to show that the static semantics and the dynamic semantics are consistent with respect to a structural relation between values and types.

2.5.1 Consistency Relations

Such a structural relation is defined in [Tofte, 1987], written $s, \mathcal{S} \models v : \tau$, for " v has type τ with the store s and the model $\mathcal{S}$". It is defined by the maximal fixed point of a monotonic property, observing that the property itself does not suffice to define the relation, because it is not well-founded by induction on values. A value v cannot always be inductively proved to have a type τ, since there might be cycles in the store s. A similar observation is made in [Milner, 1991] regarding recursive functions.

In [Leroy, 1992], the author presents a well-founded yet stronger relation. It is presented by the conjunction of $\mathcal{S} \models v : \tau$, which says that "$v$ has type τ with the model $\mathcal{S}$" and $\models s : \mathcal{S}$, which says that "the store s has model $\mathcal{S}$". This representation elegantly separates the predicates of value typing and store typing. However, it is based on a very simplified representation of recursive functions which does not permit to represent mutually recursive functions.

In the following, we adopt an alternate predicate, which is inspired from [Leroy, 1992], for describing store typings, and from [Milner, 1991], for describing the typing of recursive functions.

Definition 2.1 (Store Model) *A store model $\mathcal{S}$, defined on StoreModel=Ref $\xrightarrow{\text{fin}}$ Region$\times$ Type, is a finite mapping from locations l to pairs (ρ, τ) of regions ρ and types τ. We say that $\mathcal{S}'$ extends $\mathcal{S}$, written $\mathcal{S} \sqsubseteq \mathcal{S}'$, if and only if $Dom(\mathcal{S}) \subseteq Dom(\mathcal{S}')$ and for every $l \in Dom(\mathcal{S})$, $\mathcal{S}(l) = \mathcal{S}'(l)$.*

The notion of store model is used to describe the relation between the objects of the dynamic and the static semantics.

Definition 2.2 (Consistent values and types) *A value v is consistent with the type τ for the model $\mathcal{S}$, written $\mathcal{S} \models v : \tau$, if and only if v and τ verify one of the following properties:*

$$\mathcal{S} \models u : unit$$
$$\mathcal{S} \models l : ref_\rho(\tau) \Leftrightarrow \mathcal{S}(l) = (\rho, \tau)$$
$$\mathcal{S} \models (\mathtt{x}, \mathtt{e}, E) : \tau \Leftrightarrow \exists\ \mathcal{E},\ \mathcal{S} \models E : \mathcal{E}\ \wedge\ \mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset$$

We write $\mathcal{S} \models E : \mathcal{E}$ if and only if $Dom(E) = Dom(\mathcal{E})$ and $\mathcal{S} \models E(\mathtt{x}) : \mathcal{E}(\mathtt{x})$ for any $\mathtt{x} \in Dom(E)$.

It is shown in [Milner, 1991] that this structural property, between values and types, does not uniquely define a relation. Because functions can be recursively defined, it must be regarded as a fixed point equation.

Example As an example, consider the recursive function `loop` defined in the example of the section 2.3 (page 28). In the dynamic semantics, its value is represented by the object $c_{\mathtt{loop}}$ defined by the recursive equation below.

$$c_{\mathtt{loop}} = (\mathtt{x}, \mathtt{(if\ \ldots)}, E_{\mathtt{f}} + \{\mathtt{f} \mapsto c_{\mathtt{loop}}\})$$

One cannot check that $c_{\mathtt{loop}}$ has type $int \xrightarrow{write(\rho) \cup read(\rho)} int$ for the store model $\mathcal{S}$ before to check that there exists a type environment $\mathcal{E}$ to type $E_{\mathtt{f}} + \{\mathtt{f} \mapsto c_{\mathtt{loop}}\}$. This cannot be done by induction on the structure of values. Thus, one must define the fixed points of the recursive property defined in the definition 2.2 and define our relation as this greatest fixed point ■

We define a function $\mathcal{F}$ whose fixed points are the relations on $\mathcal{R}$ that verify the property defined above.

Definition 2.3 (F) *The function $\mathcal{F}$ is defined on $\mathcal{P}(\mathcal{R}) \rightarrow \mathcal{P}(\mathcal{R})$ where $\mathcal{R}$ is the domain $\mathcal{R} =$ StoreModel$\times$ Value $\times$ Type.*

$$\begin{aligned}\mathcal{F}(\mathcal{Q}) = \{(\mathcal{S}, v, \tau) \mid\ & \text{if } v = u \text{ then } \tau = unit \\ & \text{if } v = l \text{ then } \mathcal{S}(l) = (\rho, \tau') \text{ and } \tau = ref_\rho(\tau') \\ & \text{if } v = (\mathtt{x}, \mathtt{e}, E) \text{ then there exists } \mathcal{E} \text{ such that } \mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset, \\ & \qquad \text{and that } (\mathcal{S}, E(\mathtt{x}), \mathcal{E}(\mathtt{x})) \in \mathcal{Q} \text{ for every } \mathtt{x} \in Dom(E) = Dom(\mathcal{E})\}\end{aligned}$$

In order to guarantee the existence of fixed points for $\mathcal{F}$, it is sufficient to show that $\mathcal{F}$ is monotonic, according to [Tarsky, 1955].

Lemma 2.2 (Monotony of $\mathcal{F}$) *If $\mathcal{Q} \subseteq \mathcal{Q}'$ then $\mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$.*

Proof Let $\mathcal{Q}$ and $\mathcal{Q}'$ be two subsets of $\mathcal{R}$ such that $\mathcal{Q} \subseteq \mathcal{Q}'$. Let q be $(\mathcal{S}, v, \tau)$ in $\mathcal{F}(Q)$. We prove that $q \in \mathcal{F}(\mathcal{Q}')$.

- If $v = u$, then $\tau = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q}')$.
- If $v = l$, then $\mathcal{S}(l) = (\rho, \tau')$ and $\tau = \mathit{ref}_\rho(\tau')$ so $q \in \mathcal{F}(\mathcal{Q}')$.
- If $v = (\mathtt{x}, \mathtt{e}, E)$ then there exists $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset$ and that $(\mathcal{S}, E(\mathtt{x}), \mathcal{E}(\mathtt{x})) \in \mathcal{Q}$ for every $\mathtt{x} \in \mathit{Dom}(E)$. Thus, $q \in \mathcal{F}(\mathcal{Q}')$ □

Among the fixed points of $\mathcal{F}$, the greatest fixed point $\mathit{gfp}(\mathcal{F}) = \cup\{\mathcal{Q} \subseteq \mathcal{R} \mid \mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})\}$ defines our relation. A set $\mathcal{Q}$ such that $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$ is called *$\mathcal{F}$-consistent* and the relation between types and values is defined by:

$$\mathcal{S} \models v : \tau \;\Leftrightarrow\; (\mathcal{S}, v, \tau) \in \mathit{gfp}(\mathcal{F})$$

Lemma 2.3 (Values and Models Extension) *If $\mathcal{S} \sqsubseteq \mathcal{S}'$ and $\mathcal{S} \models v : \tau$ then $\mathcal{S}' \models v : \tau$.*

Proof This property is an immediate consequence of the definition 2.2. To prove it, we use the technique of co-induction, proposed in [Tofte, 1987]. It consist to prove that the set $\mathcal{Q} = \{(\mathcal{S}', v, \tau) \mid \mathcal{S} \models v : \tau\}$, where $\mathcal{S}'$ extends $\mathcal{S}$, is $\mathcal{F}$-consistent. To do this, it is sufficient to show that $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$. Let $q = (\mathcal{S}', v, \tau) \in \mathcal{Q}$, we show that $q \in \mathcal{F}(\mathcal{Q})$ by case analysis on the structure of values.

- If $v = u$ then, by definition of $\models$, $\tau = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.
- If $v = l$ then, by definition of $\models$, $\tau = \mathit{ref}_\rho(\tau)$ and $\mathcal{S}(l) = (\rho, \tau')$. Since $\mathcal{S} \subseteq \mathcal{S}'$, one has $\mathcal{S}'(l) = (\rho, \tau')$ so that $q \in \mathcal{F}(\mathcal{Q})$.
- If $v = (\mathtt{x}, \mathtt{e}, E)$ then, by definition of $\models$, there exists $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset$ and that $\mathcal{S} \models E(\mathtt{x}){:}\mathcal{E}(\mathtt{x})$ for every $\mathtt{x} \in \mathit{Dom}(E)$. By definition of $\mathcal{Q}$, $(\mathcal{S}', E(\mathtt{x}), \mathcal{E}(\mathtt{x})) \in \mathcal{Q}$. Thus, $q \in \mathcal{F}(\mathcal{Q})$ □

Lemma 2.4 (Values and Semantics Substitution) *If $\mathcal{S} \models v : \tau$ then $\theta\mathcal{S} \models v : \theta\tau$ for any θ.*

Proof The proof is by co-induction. Let us consider the set $\mathcal{Q} = \{(\theta\mathcal{S}, v, \theta\tau) \mid \mathcal{S} \models v : \tau\}$. To prove that $\mathcal{Q}$ is $\mathcal{F}$-consistent, it is sufficient to show that $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$. Let $q = (\theta\mathcal{S}, v, \theta\tau) \in \mathcal{Q}$, we show that $q \in \mathcal{F}(\mathcal{Q})$ by case analysis on the structure of values.

- If $v = u$ then, by definition of $\models$, $\tau = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.
- If $v = l$ then $\tau = \mathit{ref}_\rho(\tau')$ and $\mathcal{S}(l) = (\rho, \tau')$. Thus $\theta\mathcal{S}(l) = (\theta\rho, \theta\tau')$ and $\theta\tau = \mathit{ref}_{\theta\rho}(\theta\tau')$ so that $q \in \mathcal{F}(\mathcal{Q})$.
- If $v = (\mathtt{x}, \mathtt{e}, E)$ then, by definition of $\models$, there exists $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset$ and $\mathcal{S} \models E(\mathtt{x}) : \mathcal{E}(\mathtt{x})$ for every $\mathtt{x} \in \mathit{Dom}(E)$. By definition of $\mathcal{Q}$, $(\theta\mathcal{S}, E(\mathtt{x}), \theta\mathcal{E}(\mathtt{x})) \in \mathcal{Q}$. By the lemma 2.1, $\theta\mathcal{E} \vdash$ `(lambda (x) e)` $: \theta\tau, \emptyset$. Thus, $q \in \mathcal{F}(\mathcal{Q})$ □

In the same manner than between values and types, we define a consistency relation between traces and effects.

Definition 2.4 (Consistent trace and effect) *A dynamic trace f is consistent with the static effect σ for the store model $\mathcal{S}$, written $\mathcal{S} \models f : \sigma$.*

$$\mathcal{S} \models f : \sigma \Leftrightarrow \begin{cases} \forall\ \mathit{init}(l) \in f, & \mathcal{S}(l) = (\rho, \tau) \ \wedge\ \mathit{init}(\rho) \in \sigma \\ \forall\ \mathit{read}(l) \in f, & \mathcal{S}(l) = (\rho, \tau) \ \wedge\ \mathit{read}(\rho) \in \sigma \\ \forall\ \mathit{write}(l) \in f, & \mathcal{S}(l) = (\rho, \tau) \ \wedge\ \mathit{write}(\rho) \in \sigma \end{cases}$$

Lemma 2.5 (Traces and Models Extension) *If $\mathcal{S} \sqsubseteq \mathcal{S}'$ and $\mathcal{S} \models f : \sigma$ then $\mathcal{S}' \models f : \sigma$. Also if $\mathcal{S} \models f : \sigma$ then $\mathcal{S} \models f : \sigma \cup \sigma'$.*

Proof These properties are immediate consequences of the definition 2.4. By hypothesis, $\mathcal{S} \sqsubseteq \mathcal{S}'$ and $\mathcal{S} \models f : \sigma$. For every $init(l)$ (respectively $read(l)$ and $write(l)$) in f, we have $\mathcal{S}(l) = (\rho, \tau)$ and $init(\rho)$ (respectively $read(\rho)$ and $write(\rho)$) in σ. Since $\mathcal{S}'$ extends $\mathcal{S}$ and by definition, $\mathcal{S}'(l) = (\rho, \tau)$. We conclude that $\mathcal{S}' \models f : \sigma$. Similarly, $init(\rho)$ (respectively $read(\rho)$ and $write(\rho)$) are in $\sigma \cup \sigma'$ and thus $\mathcal{S} \models f : \sigma \cup \sigma'$ □

Lemma 2.6 (Traces and Semantics Substitution) *If $\mathcal{S} \models f : \sigma$ then $\theta\mathcal{S} \models f : \theta\sigma$ for any θ.*

Proof This property is an immediate consequence of the definition 2.4. By hypothesis, $\mathcal{S} \models f : \sigma$. By definition, $\mathcal{S}(l) = (\rho, \tau)$ and $init(\rho)$ (respectively $read(\rho)$ and $write(\rho)$) is in σ for every l such that $init(l)$ (respectively $read(l)$ and $write(l)$) is in f. For any substitution θ, we conclude that $\theta\mathcal{S} \models f : \theta\sigma$ by definition □

Now, we define the relation between the dynamic stores s and the semantics models $\mathcal{S}$, in the same way than in [Leroy, 1992].

Definition 2.5 (Consistent store) *A store s is consistent with the model $\mathcal{S}$, written $\models s : \mathcal{S}$, if and only if $Dom(\mathcal{S}) = Dom(s)$ and for every $l \in Dom(s)$, $\mathcal{S}(l) = (\rho, \tau)$ and $\mathcal{S} \models s(l) : \tau$.*

Lemma 2.7 (Stores and Semantics Substitution) *If $\models s : \mathcal{S}$ then $\models s : \theta\mathcal{S}$ for any θ.*

Proof This property is an immediate consequences of the definition 2.5. By hypothesis, $\models s : \mathcal{S}$. By definition, $Dom(\mathcal{S}) = Dom(s)$, $\mathcal{S}(l) = (\rho, \tau)$ and $\mathcal{S} \models s(l) : \tau$ for every $l \in Dom(s)$. By the lemma 2.4, we have $\theta\mathcal{S} \models s(l) : \theta\tau$ where $\theta\mathcal{S}(l) = (\theta\rho, \theta\tau)$. By definition, we conclude that $\models s : \theta\mathcal{S}$ □

2.5.2 Consistency Theorem

We are now going to state the theorem of consistency between the dynamic and the static semantics. It states that the value v and the type τ of an expression `e` are consistent. It does not state that there always exists a value v to which `e` evaluates, since a well typed program may fail to terminate. However, the aim of a type system for a programming language is not to ensure termination, but to guarantee the structural conformity of data.

Theorem 2.1 (Consistency of dynamic and static semantics) *If $\mathcal{S} \models E : \mathcal{E}$, $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ and $s, E \vdash \mathtt{e} \rightarrow v, f, s'$ then there exists a store model $\mathcal{S}'$ extending $\mathcal{S}$ such that $\mathcal{S}' \models v : \tau$, $\mathcal{S}' \models f : \sigma$ and $\models s' : \mathcal{S}'$.*

Proof The proof is by induction on the derivation in the dynamic semantics.

Case of (var) By hypothesis $\mathcal{S} \models E : \mathcal{E}$, $\mathcal{E} \vdash \mathtt{e} : \mathcal{E}(\mathtt{x}), \emptyset$ and $s, E \vdash \mathtt{x} \rightarrow E(\mathtt{x}), \emptyset, s$. By definition of the rule (var), this requires that $\mathtt{x} \in Dom(E)$ and $\mathtt{x} \in Dom(\mathcal{E})$. Since $\mathcal{S} \models E : \mathcal{E}$ and taking $\mathcal{S}' = \mathcal{S}$, we conclude that $\mathcal{S}' \models v : \tau$, $\mathcal{S}' \models \emptyset : \emptyset$ and $\models s' : \mathcal{S}'$ (where $s' = s$).

Case of (abs) By hypothesis $\mathcal{S} \models E : \mathcal{E}$, $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset$ and $s, E \vdash$ `(lambda (x) e)` $\rightarrow (\mathtt{x}, \mathtt{e}, E_{\mathtt{x}}), \emptyset, s$. Taking $\mathcal{S}' = \mathcal{S}$, we conclude that $\mathcal{S}' \models (\mathtt{x}, \mathtt{e}, E_{\mathtt{x}}) : \tau$, $\mathcal{S}' \models \emptyset : \emptyset$ and $\models s' : \mathcal{S}'$ (where $s' = s$).

Case of (rec) By hypothesis, $\mathcal{S} \models E : \mathcal{E}$, $\mathcal{E} \vdash$ `(rec (f x) e)` $: \tau, \emptyset$ and $s, E \vdash$ `(rec (f x) e)` $\rightarrow c, \emptyset, s$ where $c = (\mathtt{x}, \mathtt{e}, E_{\mathtt{x},\mathtt{f}} + \{\mathtt{f} \mapsto c\})$. Let us write $\mathcal{E}' = \mathcal{E}_{\mathtt{f}} + \{\mathtt{f} \mapsto \tau\}$. By definition of the rule (rec), the hypothesis requires that $\mathcal{E}' \vdash$ `(lambda (x) e)` $: \tau, \emptyset$.

Take $E' = E_{\mathtt{f}} + \{\mathtt{f} \mapsto c\}$, $\mathcal{S}' = \mathcal{S}$ and $s' = s$. To prove $\mathcal{S}' \models c : \tau$, it is sufficient to show that $(\mathcal{S}, c, \tau) \in gfp(\mathcal{F})$. To this end, we define $\mathcal{Q} = gfp(\mathcal{F}) \cup \{(\mathcal{S}, c, \tau)\}$ and show that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

Consider any $q \in \mathcal{Q}$. If $q \in gfp(\mathcal{F})$ then, since $gfp(\mathcal{F}) \subseteq \mathcal{Q}$ and $\mathcal{F}$ is monotonic, $q \in \mathcal{F}(\mathcal{Q})$. Otherwise, $q = (\mathcal{S}, c, \tau)$. Since $\mathcal{E}' \vdash$ `(lambda (x) e)` $: \tau, \emptyset$, and $(\mathcal{S}, E(\mathtt{y}), \mathcal{E}(\mathtt{y})) \in \mathcal{Q}$ for every $\mathtt{y} \in Dom(E)$, $q \in \mathcal{Q}$. We get that $(\mathcal{S}, E'(\mathtt{y}), \mathcal{E}'(\mathtt{y})) \in \mathcal{Q}$ for every $\mathtt{y} \in Dom(\mathcal{E})$. This proves that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

We conclude that $\mathcal{S} \models c : \tau$.

Case of (app) By hypothesis $\mathcal{S} \models E : \mathcal{E},\ \mathcal{E} \vdash (\mathtt{e}\ \mathtt{e}') : \tau', \sigma \cup \sigma' \cup \sigma''$ and $s, E \vdash (\mathtt{e}\ \mathtt{e}') \rightarrow v', f \cup f' \cup f'', s'$. By the definition of rule *(app)*, we have that $\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\sigma''} \tau', \sigma$ and that $\mathcal{E} \vdash \mathtt{e}' : \tau, \sigma'$. By definition of the rule (app) in the dynamic semantics, we have that

$$s, E \vdash \mathtt{e} \rightarrow (\mathtt{x}, \mathtt{e}'', E'), f, s_1 \qquad s_1, E \vdash \mathtt{e}' \rightarrow v, f', s_2 \quad \text{and} \quad s_2, E' + \{\mathtt{x} \mapsto v\} \vdash \mathtt{e}'' \rightarrow v', f'', s'$$

By induction on $\mathtt{e}$, there exists a store model $\mathcal{S}_1$ extending $\mathcal{S}$ such that

$$\mathcal{S}_1 \models (\mathtt{x}, \mathtt{e}'', E') : \tau \xrightarrow{\sigma''} \tau' \qquad \mathcal{S}_1 \models f : \sigma \quad \text{and} \quad \models s_1 : \mathcal{S}_1$$

Since $\mathcal{S}_1$ extends $\mathcal{S}$ and $\mathcal{S} \models E : \mathcal{E}$, then by the lemma 2.3, $\mathcal{S}_1 \models E : \mathcal{E}$. By induction on $\mathtt{e}'$, there exists a store model $\mathcal{S}_2$ extending $\mathcal{S}_1$ such that

$$\mathcal{S}_2 \models v : \tau \qquad \mathcal{S}_2 \models f' : \sigma' \quad \text{and} \quad \models s_2 : \mathcal{S}_2$$

Since $\mathcal{S}_2$ extends $\mathcal{S}_1$, since $\mathcal{S}_1 \models E : \mathcal{E}$ and $\mathcal{S}_1 \models (\mathtt{x}, \mathtt{e}'', E') : \tau \xrightarrow{\sigma''} \tau'$ and by the lemma 2.3, we have that

$$\mathcal{S}_2 \models E : \mathcal{E} \quad \text{and} \quad \mathcal{S}_2 \models (\mathtt{x}, \mathtt{e}'', E') : \tau \xrightarrow{\sigma''} \tau'$$

By definition of the relation $\models$, there exists a type environment $\mathcal{E}'$ such that $\mathcal{S}_2 \models E' : \mathcal{E}'$. Since $\mathcal{S}_2 \models v : \tau$ and by definition $\mathcal{S}_2 \models E' + \{\mathtt{x} \mapsto v\} : \mathcal{E}' + \{\mathtt{x} \mapsto \tau\}$. By induction hypothesis on $\mathtt{e}''$, there exists a model $\mathcal{S}'$ extending $\mathcal{S}_2$ such that

$$\mathcal{S}' \models f'' : \sigma'' \qquad s' : \mathcal{S}' \models v' : \tau' \quad \text{and} \quad \models s' : \mathcal{S}'$$

This allows us to conclude that $\mathcal{S}'$ extends $\mathcal{S}$ and verifies

$$s' : \mathcal{S}' \models v' : \tau' \qquad \mathcal{S}' \models f \cup f' \cup f'' : \sigma \cup \sigma' \cup \sigma'' \quad \text{and} \quad \models s' : \mathcal{S}'$$

Case of (new) By hypothesis $\mathcal{S} \models E : \mathcal{E},\ \mathcal{E} \vdash (\mathtt{new}\ \mathtt{e}) : \mathit{ref}_\rho(\tau), \sigma \cup \mathit{init}(\rho)$ and $s, E \vdash (\mathtt{new}\ \mathtt{e}) \rightarrow l, f \cup \{\mathit{init}(l)\}, s' + \{l \mapsto v\}$ where $l \notin \mathit{Dom}(s')$. By definition of the static semantics, this requires that $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$. By definition of the dynamic semantics, this requires that $s, E \vdash \mathtt{e} \rightarrow v, f, s'$. By induction on $\mathtt{e}$, there exists a store model $\mathcal{S}_1$ extending $\mathcal{S}$ such that

$$\mathcal{S}_1 \models v : \tau \qquad \mathcal{S}_1 \models f : \sigma \quad \text{and} \quad \models s' : \mathcal{S}_1$$

Let us define $\mathcal{S}'$ by $\mathcal{S}_1 + \{l \mapsto (\rho, \tau)\}$. Since $l \notin \mathit{Dom}(s')$, we have that $\models s' + \{l \mapsto v\} : \mathcal{S}'$ and that $\mathcal{S}' \models f \cup \{\mathit{init}(l)\} : \sigma \cup \mathit{init}(\rho)$. We conclude that there exists $\mathcal{S}'$ extending $\mathcal{S}$ such that

$$\mathcal{S}' \models v : \tau \qquad \mathcal{S}' \models f \cup \{\mathit{init}(l)\} : \sigma \cup \mathit{init}(\rho) \quad \text{and} \quad \models s' + \{l \mapsto v\} : \mathcal{S}'$$

Case of (get) By hypothesis $\mathcal{S} \models E : \mathcal{E},\ \mathcal{E} \vdash (\mathtt{get}\ \mathtt{e}) : \tau, \sigma \cup \mathit{read}(\rho)$ and $s, E \vdash (\mathtt{get}\ \mathtt{e}) \rightarrow s(l), f \cup \{\mathit{read}(l)\}, s'$. This requires that $s, E \vdash \mathtt{e} \rightarrow l, f, s'$ and $\mathcal{E} \vdash \mathtt{e} : \mathit{ref}_\rho(\tau), \sigma$. By induction hypothesis on $\mathtt{e}$, there exists $\mathcal{S}'$ extending $\mathcal{S}$ such that

$$\mathcal{S}' \models l : \mathit{ref}_\rho(\tau) \qquad \mathcal{S}' \models f : \sigma \quad \text{and} \quad \models s' : \mathcal{S}'$$

Since $\mathcal{S}' \models l : \mathit{ref}_\rho(\tau)$ and by definition of $\models$, $\mathcal{S}(l) = (\rho, \tau)$. Thus, and since $\mathcal{S}' \models f : \sigma$, we have that $\mathcal{S}' \models f \cup \{\mathit{read}(l)\} : \sigma \cup \mathit{read}(\rho)$. Since $s'(l) = v$ and $\mathcal{S}(l) = (\rho, \tau)$, we conclude that $\mathcal{S}'$ extends $\mathcal{S}$ and that

$$\mathcal{S}' \models v : \tau \qquad \mathcal{S}' \models f \cup \{\mathit{read}(l)\} : \sigma \cup \mathit{read}(\rho) \quad \text{and} \quad \models s' : \mathcal{S}'$$

Case of (set) By hypothesis $\mathcal{S} \models E : \mathcal{E}$, $\mathcal{E} \vdash$ `((set e) e')` $: unit, \sigma \cup \sigma' \cup write(\rho)$ and $s, E \vdash$ `((set e) e')` $\rightarrow u, f \cup f' \cup \{write(l)\}, s'$. In the dynamic semantics, this requires that $s, E \vdash$ `e` $\rightarrow l, f, s_1$ and $s_1, E \vdash$ `e'` $\rightarrow v, f', s_2$ and $s' = s_{2l} + \{l \mapsto v\}$. In the static semantics, we must have $\mathcal{E} \vdash$ `e` $: ref_\rho(\tau), \sigma$ and $\mathcal{E} \vdash$ `e'` $: \tau, \sigma'$. By induction hypothesis on `e`, there exists a model $\mathcal{S}_1$ extending $\mathcal{S}$ such that

$$\mathcal{S}_1 \models f : \sigma \qquad \mathcal{S}_1 \models l : ref_\rho(\tau) \quad \text{and} \quad \models s_1 : \mathcal{S}_1$$

Since $\mathcal{S}_1$ extends $\mathcal{S}$ and $\mathcal{S} \models E : \mathcal{E}$ then $\mathcal{S}_1 \models E : \mathcal{E}$. By induction hypothesis on `e'`, there exists $\mathcal{S}'$ extending $\mathcal{S}_1$ such that

$$\mathcal{S}' \models v : \tau \qquad \mathcal{S}' \models f' : \sigma' \quad \text{and} \quad \models s_2 : \mathcal{S}'$$

Since $\mathcal{S}_1 \models l : ref_\rho(\tau)$ and by definition of $\models$, $\mathcal{S}_1(l) = (\rho, \tau)$. Since $\mathcal{S}'$ extends $\mathcal{S}_1$, $s' = s_{2l} + \{l \mapsto v\}$, $\mathcal{S}_1 \models f : \sigma$ and $\mathcal{S}' \models f' : \sigma'$, we conclude

$$\mathcal{S}' \models u : unit \qquad \mathcal{S}' \models f \cup f' \cup \{write(l)\} : \sigma \cup \sigma' \cup write(\rho) \quad \text{and} \quad \models s' : \mathcal{S}' \ \square$$

2.6 Reconstruction Algorithm

In this section, we present the algorithm $\mathcal{I}$ for reconstructing the type and effect of expressions. We discuss the central ideas of our approach, describe the unification process, give the reconstruction algorithm and discuss its properties.

2.6.1 Presentation

Given a type environment and an expression, the algorithm $\mathcal{I}$ reconstructs the principal type and effect of an expression which can be inferred using the axioms and rules defined in section 2.3. The algorithm $\mathcal{I}$ resembles to Damas & Milner's algorithm W [Damas, 1985]. However, if the idea of Milner's algorithm W had been applied naively, designing the unification algorithm so as to perform the matching of effects occurring on the arrow of function types would have been problematic. The rule (does) of the static semantics and the algorithm are designed so to overcome this problem.

Just as for algorithm W, the reconstruction of the type and effect of expressions is a constraint satisfaction problem. In accordance with the static semantics, the algorithm computes equations between types, equations between regions, and inequations between effects. Just like for the Damas & Milner algorithm W, the main invariant of the algorithm is that an expression admits a type and an effect in the static semantics if, and only if, the system of equations and inequations, collected by the algorithm $\mathcal{I}$, has at least one solution.

Our algorithm $\mathcal{I}$ has however a particular invariant. It represents the function types $\tau \xrightarrow{\sigma} \tau'$ of the static semantics by a type $\tau \xrightarrow{\varsigma} \tau$ and a constraint $\varsigma \supseteq \sigma$, where ς is an effect variable. This makes the problem of solving equations on types amenable to first-order term unification [Robinson, 1965]. As in Damas & Milner's algorithm W, a call $\mathcal{U}(\tau, \tau')$ to the unification algorithm either fails or returns a most general substitution θ unifying τ and τ'.

2.6.2 Constraint Sets

Formally, a *constraint* $\varsigma \supseteq \sigma$ is a pair consisting of an effect variable ς and an effect σ. Constraints are collected into constraint sets which are written κ. An inequation $\varsigma \supseteq \sigma$ in κ enforces to assign a lower bound of value σ to the inferred effect variable ς in order to be consistent with the static semantics.

Constraints are introduced by the algorithm $\mathcal{I}$ when it processes `lambda` and `rec` expressions which is the place where effects are introduced into types. We prove below that, by construction, these constraint sets *always* admit at least one solution. The notion of solution, or model of constraint sets is formally defined as follows:

Definition 2.6 (Model of Constraints) *A substitution θ models a constraint set κ, written $\theta \models \kappa$, if and only if $\theta\varsigma \supseteq \theta\sigma$ for every constraint $\varsigma \supseteq \sigma$ in κ. The principal model $\overline{\kappa}$ of κ is inductively defined by:*

$$\overline{\emptyset} = Id \text{ and } \overline{\kappa \cup \{\varsigma \supseteq \sigma'\}} = \{\varsigma \mapsto \sigma'\} \circ \overline{\kappa} \text{ where } \sigma' = \overline{\kappa}(\varsigma \cup \sigma)$$

2.6.3 The Reconstruction Algorithm

Given a type environment $\mathcal{E}$ and an expression $\mathtt{e}$, the reconstruction algorithm $\mathcal{I}$ computes a substitution θ ranging over the free type, effect and region variables of the type environment $\mathcal{E}$, a type τ, an effect σ and a constraint set κ containing all the inequations that need to be satisfied by effect variables in order to preserve the static semantics.

$$
\begin{array}{rl}
\mathcal{I}(\mathcal{E},\mathtt{e}) = & \text{case } \mathtt{e} \text{ of} \\
\mathtt{set} \Rightarrow & \text{let } \alpha,\ \varrho,\ \varsigma,\ \varsigma' \text{ fresh in } (\mathit{Id}, \mathit{ref}_\varrho(\alpha) \xrightarrow{\varsigma} \alpha \xrightarrow{\varsigma'} \mathit{unit}, \emptyset, \{\varsigma' \supseteq \mathit{write}(\varrho)\}) \\
\mathtt{get} \Rightarrow & \text{let } \alpha,\ \varrho,\ \varsigma \text{ fresh in } (\mathit{Id}, \mathit{ref}_\varrho(\alpha) \xrightarrow{\varsigma} \alpha, \emptyset, \{\varsigma \supseteq \mathit{read}(\varrho)\}) \\
\mathtt{new} \Rightarrow & \text{let } \alpha,\ \varrho,\ \varsigma \text{ fresh in } (\mathit{Id}, \alpha \xrightarrow{\varsigma} \mathit{ref}_\varrho(\alpha), \emptyset, \{\varsigma \supseteq \mathit{init}(\varrho)\}) \\
\mathtt{x} \Rightarrow & \text{if } \mathtt{x} \in \mathit{Dom}(\mathcal{E}) \text{ then let } (\tau,\kappa) = \mathit{Inst}(\mathcal{E}(\mathtt{x})) \text{ in } (\mathit{Id}, \tau, \emptyset, \kappa) \text{ else fail} \\
\mathtt{(lambda\ (x)\ e)} \Rightarrow & \text{let } \alpha, \varsigma \text{ new and } (\theta,\tau,\sigma,\kappa) = \mathcal{I}(\mathcal{E}_\mathtt{x} + \{\mathtt{x} \mapsto \alpha\}, \mathtt{e}) \\
& \text{in } (\theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa \cup \{\varsigma \supseteq \sigma\}) \\
\mathtt{(rec\ (f\ x)\ e)} \Rightarrow & \text{let } \alpha, (\theta,\tau,\sigma,\kappa) = \mathcal{I}(\mathcal{E}_\mathtt{f} + \{\mathtt{f} \mapsto \alpha\}, \mathtt{(lambda\ (x)\ e)}) \text{ and } \theta' = \mathcal{U}(\theta\alpha, \tau) \\
& \text{in } (\theta' \circ \theta, \theta'\tau, \emptyset, \theta'\kappa) \\
\mathtt{(e\ e')} \Rightarrow & \text{let } (\theta,\tau,\sigma,\kappa) = \mathcal{I}(\mathcal{E},\mathtt{e}),\ (\theta',\tau',\sigma',\kappa') = \mathcal{I}(\theta\mathcal{E},\mathtt{e}'),\ \alpha,\ \varsigma \text{ new and } \theta'' = \mathcal{U}(\theta'\tau, \tau' \xrightarrow{\varsigma} \alpha) \\
& \text{in } (\theta'' \circ \theta' \circ \theta, \theta''\alpha, \theta''(\theta'\sigma \cup \sigma' \cup \varsigma), \theta''(\theta'\kappa \cup \kappa'))
\end{array}
$$

Reconstruction Algorithm $\mathcal{I}$

The algorithm $\mathcal{I}$ is defined by a recursive function which works by case analysis on the syntax of the expression $\mathtt{e}$ supplied to it. In the case of an operator `op`, the type and constraint set of that operator, instantiated with the procedure *Inst*, are returned. In the case of an identifier `x`, it is first checked that the identifier is defined. Then, the constrained type scheme associated with it in the type environment is instantiated and returned.

In the case of a lambda-abstraction `(lambda (x) e)`, the body `e` is checked with the assumption that `x` has type α. When this is done, the substitution θ holds the appropriate type to substitute to α. A function type $\theta\alpha \xrightarrow{\varsigma} \tau$ is then built with a fresh ς. In the constraint set which is returned, ς is associated with the lower bound of the effect of the lambda-abstraction, σ, which was computed when its body `e` was checked.

In the case analysis for the application expression, the subexpressions `e` and `e'` are first checked and then, it is checked, using the unification algorithm, that the type of the argument `e'`, supplied to the function `e`, corresponds to the type of the formal parameter of the function. The type of the result of the function is returned, together with the sum of substitutions, effects and constraint sets computed for the subexpressions of the application.

2.6.4 The Unification Algorithm

Equations between types, between effect variables and between regions are solved by a Robinson-like unification algorithm $\mathcal{U}$. The algorithm $\mathcal{I}$ always calls $\mathcal{U}$ with two terms of a free algebra.

$$
\begin{array}{rcl}
\mathcal{U}(\tau,\tau') & = & \text{case } (\tau,\tau') \text{ of} \\
(\mathit{unit}, \mathit{unit}) & \Rightarrow & \mathit{Id} \\
(\alpha,\alpha') & \Rightarrow & \{\alpha \mapsto \alpha'\} \\
(\alpha,\tau) | (\tau,\alpha) & \Rightarrow & \text{if } \alpha \in \mathit{fv}(\tau) \text{ then fail else } \{\alpha \mapsto \tau\} \\
(\tau_i \xrightarrow{\varsigma} \tau_f, \tau_i' \xrightarrow{\varsigma'} \tau_f') & \Rightarrow & \text{let } \theta = \{\varsigma \mapsto \varsigma'\} \text{ and } \theta' = \mathcal{U}(\theta\tau_i, \theta\tau_i') \text{ in } \mathcal{U}(\theta'(\theta\tau_f), \theta'(\theta\tau_f')) \circ \theta' \circ \theta \\
(\mathit{ref}_\varrho(\tau), \mathit{ref}_{\varrho'}(\tau')) & \Rightarrow & \text{let } \theta = \{\varrho \mapsto \varrho'\} \text{ in } \mathcal{U}(\theta\tau, \theta\tau') \circ \theta \\
\text{otherwise} & \Rightarrow & \text{fail}
\end{array}
$$

Unification Algorithm

A call $\mathcal{U}(\tau,\tau')$ to the unification algorithm $\mathcal{U}$ either fails or returns a most general unifying substitution θ of τ and τ' defined on the free variables of τ and τ'. A substitution θ is an unifier of two types τ and τ' if $\theta\tau = \theta\tau'$. Two types are said unifiable if there exists a unifier for them. A unifier θ for τ and τ' is most general if all unifiers θ' for τ and τ' are such that $\theta' = \theta'' \circ \theta$ for some substitution θ''.

The algorithm $\mathcal{U}$ proceeds by case analysis on the structure of the type terms supplied to it, and is recursively called on every non terminal subterms. The case of matching basic data types is trivial. In the case of two variables α and α', one is substituted for the other. The case of unifying a variable α with a type τ is less simple. The solution $\theta = \{\alpha \mapsto \tau\}$ must satisfy the equation $\theta\alpha = \theta\tau$. This cannot be the case if the variable α is free in τ. In this case, the solution of α is recursively defined. It is an infinite term. This is why it is here first checked that $\alpha \notin fv(\tau)$. The case of unifying data structures and functions reduces to the composing the substitutions obtained by checking the subterms of the type supplied using recursive calls to the unification procedure.

The formal properties of our unification algorithm are stated by the following theorem. Unification with $\mathcal{U}$ is sound in that, if $\mathcal{U}(\tau,\tau')$ returns a substitution θ, then θ unifies τ and τ'. Unification with $\mathcal{U}$ is complete in that it always returns the most general unifier of two terms.

Lemma 2.8 (Correctness of $\mathcal{U}$) *Let τ and τ' be two type terms in the domain of $\mathcal{U}$. If $\mathcal{U}(\tau,\tau') = \theta$, then $\theta\tau = \theta\tau'$ and, whenever $\theta'\tau = \theta'\tau'$, there exists a substitution θ'' such that $\theta' = \theta'' \circ \theta$.*

Proof The proof is by induction on terms as in [Robinson, 1965] □

2.7 Formal Properties of Constraint Sets

The formal properties of the principal model $\overline{\kappa}$ of a constraint set κ can be stated by the two following lemmas, which tell that every constraint set κ admits the principal model $\overline{\kappa}$ as a solution and that every solution θ of a constraint set κ can be expressed with the principal model $\overline{\kappa}$.

Lemma 2.9 (Principal Model) *For all κ, the substitution $\overline{\kappa}$ solves κ.*

Proof The proof is by induction on the number of constraints in κ. If $\kappa = \emptyset$, then $\overline{\kappa} = Id$ solves κ. Consider $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$ where $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$. By definition, we have that $\overline{\kappa} = \{\varsigma \mapsto \overline{\kappa'}(\varsigma \cup \sigma)\} \circ \overline{\kappa'}$ and by induction hypothesis on κ', we have that $\overline{\kappa'}$ solves κ'. For every constraint $\varsigma' \supseteq \sigma'$ in κ', $\overline{\kappa}(\varsigma') = \{\varsigma \mapsto \overline{\kappa'}(\varsigma \cup \sigma)\}(\overline{\kappa'}(\varsigma'))$ and $\overline{\kappa}\sigma' = \{\varsigma \mapsto \overline{\kappa'}(\varsigma \cup \sigma)\}(\overline{\kappa'}(\sigma'))$.

- If $\varsigma \in \overline{\kappa'}(\varsigma')$ then $\overline{\kappa}(\varsigma') = (\overline{\kappa'}(\varsigma') \setminus \varsigma) \cup \overline{\kappa'}(\varsigma \cup \sigma)$. Since $\varsigma \in \overline{\kappa'}(\varsigma)$ we have $\overline{\kappa}(\varsigma') = \overline{\kappa'}(\varsigma') \cup \overline{\kappa'}(\varsigma \cup \sigma)$. By induction we have $\overline{\kappa'}(\varsigma') \supseteq \overline{\kappa'}(\sigma')$. If $\varsigma \in \overline{\kappa'}(\sigma')$ then $\overline{\kappa}(\sigma') = (\overline{\kappa'}(\sigma') \setminus \varsigma) \cup \overline{\kappa'}(\varsigma \cup \sigma)$ so that $\overline{\kappa}(\varsigma') \supseteq \overline{\kappa}(\sigma')$. Otherwise $\varsigma \notin \overline{\kappa'}(\sigma')$ so $\overline{\kappa}(\sigma') = \overline{\kappa'}(\sigma')$; thus $\overline{\kappa}(\varsigma') = \overline{\kappa'}(\varsigma') \cup \overline{\kappa'}(\varsigma \cup \sigma) \supseteq \overline{\kappa'}(\varsigma') \supseteq \overline{\kappa'}(\sigma') = \overline{\kappa}(\sigma')$
- Otherwise $\varsigma \notin \overline{\kappa'}(\varsigma')$ and $\overline{\kappa}(\varsigma') = \overline{\kappa'}(\varsigma')$. Since $\varsigma \notin \overline{\kappa'}(\varsigma')$ and $\overline{\kappa'}(\varsigma') \supseteq \overline{\kappa'}(\sigma')$ we have $\varsigma \notin \overline{\kappa'}(\sigma')$, so $\overline{\kappa}(\sigma') = \overline{\kappa'}(\sigma')$. Since $\overline{\kappa'}$ solves κ', we have that $\overline{\kappa'}(\varsigma') \supseteq \overline{\kappa'}(\sigma')$ so that $\overline{\kappa}(\varsigma') \supseteq \overline{\kappa}(\sigma')$.

It follows that $\overline{\kappa}\varsigma' \supseteq \overline{\kappa}\sigma'$. This holds for every constraint $\varsigma' \supseteq \sigma'$ in κ', so $\overline{\kappa}$ solves κ'. It remains to show that $\overline{\kappa}$ solves $\{\varsigma \supseteq \sigma\}$. By definition $\overline{\kappa}(\varsigma) = \{\varsigma \mapsto \overline{\kappa'}(\varsigma \cup \sigma)\}(\overline{\kappa'}(\varsigma))$. Since $\varsigma \in \overline{\kappa'}(\varsigma)$, $\overline{\kappa}(\varsigma) = (\overline{\kappa'}(\varsigma) \setminus \varsigma) \cup \overline{\kappa'}(\varsigma \cup \sigma) = \overline{\kappa'}(\varsigma) \cup \overline{\kappa'}(\sigma)$. Also, $\overline{\kappa}(\sigma) = \{\varsigma \mapsto \overline{\kappa'}(\varsigma \cup \sigma)\}(\overline{\kappa'}(\sigma))$. If $\varsigma \in \overline{\kappa'}(\sigma)$ then $\overline{\kappa}(\sigma) = \overline{\kappa'}(\varsigma) \cup \overline{\kappa'}(\sigma)$ otherwise $\overline{\kappa}(\sigma) = \overline{\kappa'}(\sigma)$. In both cases, we have that $\overline{\kappa}$ solves $\{\varsigma \supseteq \sigma\}$. We have thus proved that $\overline{\kappa}$ solves κ □

Lemma 2.10 (Principal Model) *If θ solves κ then $\theta = \theta \circ \overline{\kappa}$.*

Proof By induction on the number of constraints in κ. If $\kappa = \emptyset$, then $\overline{\kappa} = Id$, so $\theta = \theta \circ \overline{\kappa}$. Now let us consider $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$ where $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$ and let θ be a solution of κ. Note that θ solves κ', so by induction, $\theta = \theta \circ \overline{\kappa'}$. Let ς' be any effect variable; we wish to show $\theta(\varsigma') = \theta(\overline{\kappa}(\varsigma'))$. By definition, $\overline{\kappa}(\varsigma') = \{\varsigma \mapsto \overline{\kappa'}(\varsigma \cup \sigma)\}(\overline{\kappa'}\varsigma')$. Then, there are two cases:

- If $\varsigma \notin \overline{\kappa'}(\varsigma')$ then $\theta(\overline{\kappa}(\varsigma')) = \theta(\overline{\kappa'}(\varsigma')) = \theta(\varsigma')$, by induction.

- Otherwise, $\varsigma \in \overline{\kappa'}(\varsigma')$, so that
$$\begin{array}{lcll} \theta(\overline{\kappa}(\varsigma')) & = & \theta(\{\varsigma \mapsto \overline{\kappa'}(\varsigma \cup \sigma)\}(\overline{\kappa'}(\varsigma'))) & \\ & = & \theta(\overline{\kappa'}(\varsigma') \setminus \varsigma) \cup \theta(\overline{\kappa'}(\varsigma \cup \sigma)) & \text{as } \varsigma \in \overline{\kappa'}(\varsigma') \\ & = & \theta(\overline{\kappa'}(\varsigma')) \cup \theta(\overline{\kappa'}(\varsigma \cup \sigma)) & \text{as } \varsigma \in \overline{\kappa'}(\varsigma) \\ & = & \theta(\varsigma') \cup \theta(\varsigma \cup \sigma) & \text{by induction} \\ & = & \theta(\varsigma') \cup \theta(\varsigma) & \text{as } \theta \text{ solves } \kappa \end{array}$$

Now $\varsigma \in \overline{\kappa'}(\varsigma')$ implies that $\theta(\varsigma) \subseteq \theta(\overline{\kappa'}(\varsigma')) = \theta(\varsigma')$. Thus $\theta(\varsigma') \cup \theta(\varsigma) = \theta(\varsigma')$. Thus $\theta(\overline{\kappa}(\varsigma')) = \theta(\varsigma')$ in this case as well □

2.8 Correctness of the Algorithm

The main correctness theorems state the soundness and the completeness of the inference algorithm. The soundness theorems states that the type τ and the effect σ computed by the algorithm $\mathcal{I}$ are provable in the static semantics modulo the principal model of the inferred constraint set κ.

Theorem 2.2 (Soundness) *If* $\mathcal{I}(\mathcal{E}, \mathtt{e}) = (\theta, \tau, \sigma, \kappa)$ *then* $\overline{\kappa}(\theta\mathcal{E}) \vdash \mathtt{e} : \overline{\kappa}\tau, \overline{\kappa}\sigma$.

Proof We proceed by case analysis on the structure of expression. We thus make the assumption that expansive and non-expansive expressions are here substituted to the appropriate form. A consequence is that the type environment $\mathcal{E}$ considered in the theorem only maps lambda-bound value identifiers to types.

Case of (var) By hypothesis, $\mathcal{I}(\mathcal{E}, \mathtt{x}) = (Id, \tau, \emptyset, \emptyset)$. By definition of the algorithm $\mathcal{I}$, we have $\mathcal{E}(\mathtt{x}) = \tau$. By definition of the rule (var), we conclude that $\overline{\kappa}\mathcal{E} \vdash \mathtt{x} : \overline{\kappa}\tau, \emptyset$ where $\kappa = \emptyset$.

Case of (abs) By hypothesis, $\mathcal{I}(\mathcal{E}, \texttt{(lambda (x) e)}) = (\theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa' \cup \{\varsigma \supseteq \sigma\})$. By definition of $\mathcal{I}$, α and ς are fresh and $\mathcal{I}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}, \mathtt{e}) = (\theta, \tau, \sigma, \kappa')$. By induction hypothesis on $\mathtt{e}$,

$$\overline{\kappa'}(\theta(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\})) \vdash \mathtt{e} : \overline{\kappa'}\tau, \overline{\kappa'}\sigma$$

Since ς is fresh and by the definition of the principal model,

$$\begin{array}{ll} \overline{\kappa' \cup \{\varsigma \supseteq \sigma\}}(\varsigma) & = \overline{\kappa'}\{\varsigma \mapsto \overline{\kappa'}(\sigma \cup \varsigma)\}(\varsigma) \\ & = \overline{\kappa'}(\sigma \cup \varsigma) \\ & = \varsigma \cup \overline{\kappa'}\sigma \end{array}$$

By definition of the rule (does), we get that

$$\overline{\kappa'}(\theta(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\})) \vdash \mathtt{e} : \overline{\kappa'}\tau, \varsigma \cup \overline{\kappa'}\sigma$$

By definition of the rule (abs), we get that

$$\overline{\kappa'}(\theta\mathcal{E}) \vdash \texttt{(lambda (x) e)} : \overline{\kappa'}(\theta\alpha) \xrightarrow{\overline{\kappa}\varsigma} \overline{\kappa'}\tau, \emptyset$$

Define $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$. Since ς is new, we have that $\overline{\kappa}(\theta\mathcal{E}) = \overline{\kappa'}(\theta\mathcal{E})$ and $\overline{\kappa}(\theta\alpha) = \overline{\kappa'}(\theta\alpha)$. We can conclude that

$$\overline{\kappa}(\theta\mathcal{E}) \vdash \texttt{(lambda (x) e)} : \overline{\kappa}(\theta\alpha \xrightarrow{\varsigma} \tau), \emptyset$$

Case of (rec) By hypothesis, $\mathcal{I}(\mathcal{E}, \texttt{(rec (f x) e)}) = (\theta' \circ \theta, \theta'\tau, \emptyset, \theta'\kappa)$. By definition of $\mathcal{I}$, $\theta' = \mathcal{U}(\theta\alpha, \tau)$ and $\mathcal{I}(\mathcal{E}_{\texttt{f}} + \{\texttt{f} \mapsto \alpha\}, \texttt{e}) = (\theta, \tau, \sigma, \kappa)$ with a fresh α. By induction hypothesis on `(lambda (x) e)`,

$$\overline{\kappa}(\theta(\mathcal{E}_{\texttt{f}} + \{\texttt{f} \mapsto \alpha\})) \vdash \texttt{(lambda (x) e)} : \overline{\kappa}\tau, \emptyset$$

From $\theta' = \mathcal{U}(\theta\alpha, \tau)$ and by the lemma 2.8, we get that $\theta'(\theta\alpha) = \theta'\tau$. Thus, by the substitution lemma 2.1,

$$\overline{\kappa}(\theta'(\theta\mathcal{E}_{\texttt{f}}) + \{\texttt{f} \mapsto \theta'\tau\}) \vdash \texttt{(lambda (x) e)} : \overline{\kappa}(\theta'\tau), \emptyset$$

By definition of the rule (rec), we get that

$$\overline{\kappa}(\theta'(\theta\mathcal{E})) \vdash \texttt{(rec (f x) e)} : \overline{\kappa}(\theta'\tau), \emptyset$$

Case of (app) By hypothesis, $\mathcal{I}(\mathcal{E}, \texttt{(e e')}) = (\theta'' \circ \theta' \circ \theta, \theta''\alpha, \theta''(\theta'\sigma \cup \sigma' \cup \varsigma), \theta''(\theta'\kappa \cup \kappa'))$. By definition of $\mathcal{I}$, this requires that $\mathcal{I}(\mathcal{E}, \texttt{e}) = (\theta, \tau, \sigma, \kappa)$, $\mathcal{I}(\theta\mathcal{E}, \texttt{e}') = (\theta', \tau', \sigma'\kappa')$ and $\theta'' = \mathcal{U}(\theta'\tau, \tau' \xrightarrow{\varsigma} \alpha)$ with fresh α and ς. By induction hypothesis on $\texttt{e}$, we get that

$$\overline{\kappa}(\theta\mathcal{E}) \vdash \texttt{e} : \overline{\kappa}\tau, \overline{\kappa}\sigma$$

Define $\kappa'' = \theta''(\theta'\kappa \cup \kappa')$. Since $\overline{\kappa}''$ satisfies $\theta''(\theta'\kappa)$, $\overline{\kappa}'' \circ \theta'' \circ \theta'$ satisfies κ, so that the substitution $\theta''' = \overline{\kappa}'' \circ \theta'' \circ \theta'$ is such that $\overline{\kappa}'' \circ \theta'' \circ \theta' = \theta''' \circ \overline{\kappa}$. By the lemma 2.1, we get

$$\overline{\kappa}''(\theta''(\theta'(\theta\mathcal{E}))) \vdash \texttt{e} : \overline{\kappa}''(\theta''(\theta'\tau)), \overline{\kappa}''(\theta''(\theta'\sigma))$$

By induction hypothesis on $\texttt{e}'$ with $\theta\mathcal{E}$, we get that

$$\overline{\kappa'}(\theta'\theta\mathcal{E}) \vdash \texttt{e}' : \overline{\kappa'}\tau', \overline{\kappa'}\sigma'$$

Since $\overline{\kappa}''$ satisfies $\theta''\kappa'$, $\overline{\kappa}'' \circ \theta''$ satisfies κ', so that the substitution $\theta''' = \overline{\kappa}'' \circ \theta''$ verifies $\overline{\kappa}'' \circ \theta'' = \theta''' \circ \overline{\kappa}'$. By the lemma 2.1, we get

$$\overline{\kappa}''(\theta''(\theta'\theta\mathcal{E})) \vdash \texttt{e}' : \overline{\kappa}''(\theta''\tau'), \overline{\kappa}''(\theta''\sigma')$$

By the lemma 2.8, $\theta''(\theta'\tau) = \theta''(\tau' \xrightarrow{\varsigma} \alpha)$. As a consequence, $\overline{\kappa}''(\theta''(\theta'(\theta\mathcal{E}))) \vdash \texttt{e} : \overline{\kappa}''(\theta''(\tau' \xrightarrow{\varsigma} \alpha)), \overline{\kappa}''(\theta'\sigma)$. By definition of the rule (app), we conclude that

$$\overline{\kappa}''(\theta''(\theta'(\theta\mathcal{E}))) \vdash \texttt{(e e')} : \overline{\kappa}''(\theta''\alpha), \overline{\kappa}''(\theta''(\theta'\sigma \cup \sigma' \cup \varsigma))$$

Case of (new) By hypothesis, $\mathcal{I}(\mathcal{E}, \texttt{new}) = (Id, \alpha \xrightarrow{\varsigma} ref_{\varrho}(\alpha), \emptyset, \{\varsigma \supseteq init(\varrho)\})$ where α, ϱ and ς are fresh. Let $\kappa = \{\varsigma \supseteq init(\varrho)\}$. By the axiom (new), $\overline{\kappa}(\theta\mathcal{E}) \vdash \texttt{new} : \overline{\kappa}(\alpha \xrightarrow{\varsigma} ref_{\varrho}(\alpha)), \emptyset$ where $\theta = Id$. The case analysis of the operators `get` and `set` are similar □

The completeness theorem states that the reconstructed type τ' and effect σ' are maximal, with respect to any inferred type τ and effect σ, for some substitution θ'' that verifies the computed constraints κ'.

Theorem 2.3 (Completeness) *If $\theta''\mathcal{E} \vdash e : \tau', \sigma'$, then $\mathcal{I}(\mathcal{E}, e) = (\theta, \tau, \sigma, \kappa)$ and there exists a substitution θ' satisfying κ' such that $\theta''\mathcal{E} = \theta'(\theta\mathcal{E})$, $\tau' = \theta'\tau$ and $\sigma' \supseteq \theta'\sigma$.*

Proof The proof is by induction on the derivation.

Case of (var) By hypothesis, $\theta\mathcal{E} \vdash \texttt{x} : \tau, \emptyset$. By definition of the rule (var), this requires that $\tau = \mathcal{E}(\texttt{x})$.By definition of the algorithm $\mathcal{I}(\mathcal{E}, \texttt{x}) = (Id, \tau, \emptyset, \{\})$.

Case of (abs) By hypothesis, $\theta''\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau_1' \xrightarrow{\sigma'} \tau_2', \emptyset$. By definition of the rule (abs), this requires that

$$\theta''\mathcal{E} + \{x \mapsto \tau_1'\} \vdash \mathtt{e} : \tau_2', \sigma'$$

Let α be fresh, this is equivalent to $\theta'' \circ \{\alpha \mapsto \tau_1'\}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \tau_2', \sigma'$. By induction hypothesis on `e`, we have $(\theta, \tau_2, \sigma, \kappa) = \mathcal{I}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}, \mathtt{e})$ and there exists a substitution θ_1' satisfying κ such that

$$\theta'' \circ \{\alpha \mapsto \tau_1'\}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}) = \theta_1'(\theta(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\})), \quad \tau_2 = \theta_1'\tau_2' \quad \text{and} \quad \sigma' = \theta_1'\sigma$$

By definition of the algorithm $\mathcal{I}$, this requires that: $\mathcal{I}(\mathcal{E}, $`(lambda (x) e)`$) = (\theta, \theta\alpha \xrightarrow{\varsigma} \tau_2, \emptyset, \kappa \cup \{\varsigma \supseteq \sigma\})$ with ς fresh. Let us define $\theta' = \theta_1'\{\varsigma \mapsto \sigma'\}$. Then θ' satisfies $\kappa \cup \{\varsigma \supseteq \sigma\}$ and is such that

$$\theta''\mathcal{E} = \theta'\theta\mathcal{E} \quad \text{and} \quad \tau_1' \xrightarrow{\sigma'} \tau_2' = \theta'(\theta\alpha \xrightarrow{\varsigma} \tau_2)$$

Case of (rec) By hypothesis $\theta\mathcal{E} \vdash$ `(rec (f x) e)` $: \tau, \emptyset$. By the rule (rec), this requires that

$$\theta(\mathcal{E}_{\mathtt{f}} + \{\mathtt{f} \mapsto \tau\}) \vdash \texttt{(lambda (x) e)} : \tau, \emptyset$$

For a fresh α,

$$\theta(\{\alpha \mapsto \tau\}\mathcal{E}_{\mathtt{f}} + \{\mathtt{f} \mapsto \alpha\}) \vdash \texttt{(lambda (x) e)} : \tau, \emptyset$$

Let us write $\mathcal{E}' = \mathcal{E}_{\mathtt{f}} + \{\mathtt{f} \mapsto \alpha\}$. By induction hypothesis on `(lambda (x) e)`, $\mathcal{I}(\mathcal{E}', \mathtt{e}) = (\theta_1', \tau', \sigma', \kappa')$ and there exists a model θ_1'' of κ' such that

$$(\theta(\{\alpha \mapsto \tau\}\mathcal{E}') = \theta_1''(\theta_1'\mathcal{E}') \quad \text{and} \quad \tau = \theta_1''\tau'$$

By the lemma 2.8, since $\tau = \theta_1''(\theta_1'\alpha) = \theta_1''\tau'$, there exists θ_2' such that $\theta_2' = \mathcal{U}(\theta_1'\alpha, \tau')$. Thus, by the definition of the algorithm $\mathcal{I}$, we get:

$$\mathcal{I}(\mathcal{E}, \texttt{(rec (f x) e)}) = (\theta_2' \circ \theta_1', \theta_2'\tau', \emptyset, \theta_2'\kappa')$$

By the lemma 2.8, there exists θ'' such that $\theta_1'' = \theta'' \circ \theta_2'$. We conclude that θ'' is a model of $\theta_2'\kappa'$ such that

$$\theta\mathcal{E} = \theta''(\theta_2'(\theta_1'\mathcal{E})) \quad \text{and} \quad \tau = \theta''(\theta_2'\tau')$$

Case of (app) By hypothesis, $\theta''\mathcal{E} \vdash$ `(e e')` $: \tau_2', \sigma' \cup \sigma_1' \cup \sigma_2'$. By definition of the rule (app), this requires that

$$\theta''\mathcal{E} \vdash \mathtt{e} : \tau_1' \xrightarrow{\sigma'} \tau_2', \sigma_1' \quad \text{and} \quad \theta''\mathcal{E} \vdash \mathtt{e}' : \tau_1', \sigma_2'$$

By induction hypothesis on `e`, we get that $(\theta_1, \tau_1, \sigma_1, \kappa_1) = \mathcal{I}(\mathcal{E}, \mathtt{e})$ and that there exists a substitution θ_1' satisfying κ_1 such that

$$\theta''\mathcal{E} = \theta_1'(\theta_1\mathcal{E}) \qquad \tau_1' \xrightarrow{\sigma'} \tau_2' = \theta_1'\tau_1 \quad \text{and} \quad \sigma_1' \supseteq \theta_1'\sigma_1$$

By induction hypothesis on `e'`, we get that $(\theta_2, \tau_2, \sigma_2, \kappa_2) = \mathcal{I}(\theta_1\mathcal{E}, \mathtt{e}')$ and that there exists a model θ_2' of κ_2 such that

$$\theta''\mathcal{E} = \theta_2'(\theta_2\theta_1\mathcal{E}) \qquad \tau_1' = \theta_2'\tau_2 \quad \text{and} \quad \sigma_2' \supseteq \theta_2'\sigma_2$$

Let α_1, ς new. Let V be the set of free variables in $\theta_2(\theta_1\mathcal{E})$, τ_2, σ_2 and κ_2. Let us take fresh α_1, ς and define θ_3' as follows:

$$\theta_3' v = \begin{cases} \theta_2' v, & v \in V \\ \tau_1', & v = \alpha_1 \\ \sigma', & v = \varsigma \\ \theta_1' v, & \text{otherwise} \end{cases}$$

By this definition, θ'_3 satisfies κ_2 and we get:

$$\theta''\mathcal{E} = \theta'_3(\theta_2(\theta_1\mathcal{E})),\ \tau'_1 \xrightarrow{\sigma'} \tau'_2 = \theta'_3(\tau_2 \xrightarrow{\varsigma} \alpha_1),\ \theta'_2\sigma_2 = \theta'_3\sigma_2$$

Now, for every v in τ_1, σ_1 and κ_1, either v is free in $\theta_1\mathcal{E}$ or v is new, by definition of $\mathcal{I}$. Then, for every such v, since $\theta'_3(\theta_2(\theta_1\mathcal{E})) = \theta'_2(\theta_2(\theta_1\mathcal{E})) = \theta'_1(\theta_1\mathcal{E})$, we have:

$$\theta'_3(\theta_2 v) = \theta'_2(\theta_2 v) = \theta'_1 v$$

Otherwise, v is fresh and thus $\theta_2 v = v$, so that $\theta'_3(\theta_2 v) = \theta'_3 v = \theta'_1 v$. We get

$$\tau'_1 \xrightarrow{\sigma'} \tau'_2 = \theta'_3(\theta_2\tau_1) \qquad \theta'_1\sigma_1 = \theta'_3(\theta_2\sigma_1) \qquad \theta'_3 \models \kappa_1$$

It follows that θ'_3 satisfies $\theta_2\kappa_1 \cup \kappa_2$. Since $\theta'_3(\theta_2\tau_1) = \theta'_3(\tau_2 \xrightarrow{\varsigma} \alpha_1)$ and by the correctness of unification, there exists a substitution θ_3 such that $\theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha_1)$. Thus,

$$\theta_3(\theta_2\tau_1) = \theta_3(\tau_2 \xrightarrow{\varsigma} \alpha_1)$$

Since θ_3 is the most general unifier of $\theta_2\tau_1$ and $\tau_2 \xrightarrow{\varsigma} \alpha_1$, there exists a substitution θ' such that $\theta'_3 = \theta' \circ \theta_3$. Let us write $\sigma = \theta_3(\theta_2\sigma_1 \cup \sigma_2 \cup \varsigma)$ and $\kappa = \theta_3(\theta_2\kappa_1 \cup \kappa_2)$. By the definition of the algorithm $\mathcal{I}$, we get that $\mathcal{I}(\mathcal{E}, (\mathrm{e}\ \mathrm{e}')) = (\theta, \theta_3\alpha_1, \sigma, \kappa)$ and that there exists a model θ' of κ verifying

$$\theta''\mathcal{E} = \theta'(\theta\mathcal{E}),\ \tau'_2 = \theta'(\theta_3\alpha_1) \quad \text{and} \quad \sigma' \cup \sigma'_1 \cup \sigma'_2 \supseteq \theta'\sigma$$

Case of (new) By hypothesis, $\theta''\mathcal{E} \vdash \texttt{new} : \tau \xrightarrow{\sigma \cup init(\rho)} ref_\rho(\tau), \emptyset$. By definition of the algorithm, $\mathcal{I}(\mathcal{E}, \texttt{new}) = (Id, \alpha \xrightarrow{\varsigma} ref_\varrho(\alpha), \emptyset, \{\varsigma \supseteq init(\varrho)\})$ where α, ϱ and ς are fresh. Let θ' be defined as

$$\theta' v = \begin{cases} \tau, & v = \alpha \\ \rho, & v = \varrho \\ \sigma \cup init(\rho), & v = \varsigma \\ \theta'' v, & \text{otherwise} \end{cases}$$

θ' satisfies $\kappa = \{\varsigma \supseteq init(\varrho)\}$. We have $\theta''\mathcal{E} = \theta'$ and $\tau \xrightarrow{\sigma \cup init(\rho)} ref_\rho(\tau) = \theta'(\alpha \xrightarrow{\varsigma} ref_\varrho(\alpha))$. The case analysis of the operators `get` and `set` are similar □

2.9 Examples

In this section, we give some examples that best illustrate the effectiveness of our algorithm for inferring effects of programs. We demonstrate its use to perform code optimizations. The chapter 4 describes these issues in more details.

Example Our first example is the `gensym` function, well-known to every Lisp programmer. Its implementation consists of combining a reference with a function. This results in so called stateful functions, which is a notion close to the one of object in object-oriented languages. There is no notion of message in our example, though. The form `(define x e)` stands for the toplevel definition of `x` by `e`.

```
(define gensym (let (sym (new 0))
                 (lambda ()
                   ((set sym) (+ (get sym) 1))
                   (get sym))))
```

The expression above allocates an integer reference, noted `sym`, with the effect $init(\rho)$ where $ref_\rho(int)$ is the type of `sym`. This reference is then captured in a closure that has no argument and returns an integer. Its latent effect is to read and write the region ρ, which is precisely the region of the integer reference captured in the closure.

The algorithm assigns `gensym` the type $unit \xrightarrow{\varsigma} int$ and the constraint set $\{\varsigma \supseteq read(\varrho) \cup write(\varrho)\}$. In the static semantics, this corresponds to its principal type ■

Example Our second example shows the use of a reference which combined with the `let` construct provides an updatable identifier (a variable). Our example describe the use of such a variable for iterating a procedure `f` over the integer from 1 to `n`. It can be defined as follows:

```
(define iterate (lambda (f n)
                  (let (i (new 0))
                    (while (< (get i) n)
                      (begin
                        (f (get i))
                        ((set i) (+ (get i) 1)))))))
```

The construct `(while e e')` evaluates `e'` while `e` is true. The construct `(begin e e')` evaluates `e` and then `e'`. The higher order function `iterate` takes a procedure `f` as argument. This procedure has type $int \xrightarrow{\varsigma} \alpha$ in that it accepts integers as arguments and produces a result of no sensible type α with an undeterminated effect ς. The function `iterate` consists of a loop that calls `f n` times. It allocates an integer reference `i` initialized to `0` and then incremented to `n`. The principal type of this function is thus:

$$\texttt{iterate} : int \xrightarrow{\sigma} \alpha \times int \xrightarrow{\sigma' \cup \sigma \cup init(\varrho) \cup read(\varrho) \cup write(\varrho)} unit$$

■

2.10 Extension to Communication Effects

The language that we presented in the section 2.2.1 does not suffice by the sole addition of operations on pointers to completely integrate all features that may characterize all aspects of "realistic" programming.

In general, "real" programs do not fit into the model that their evaluation does not interfere with the rest of the "world". They usually interact with other agents, other servers, in the operating system: the file system, the windowing system and other resources. We aim here at considering programs as several expressions that may evaluate concurrently and exchange information with each other.

Such features may be required for algorithms requiring complex interleavings, frequent replications, or important repetitions of computations. Such features may provide a faster execution of programs with the availability of multiple processing units. We would therefore like to describe such interactions in our language by presenting its formalization.

In the present section, we describe this possibility by means of simple primitive operations that introduce communications and concurrency in our core language. Concurrency is provided by the addition of simple and general language constructs that permit the interleaving of computation and the non-deterministic choice of evaluation.

Communication is provided by first-class values: *channels*, which enable us to represent various protocols of interaction between independent processes. With communication channels, operations are provided to allocate and manipulate them.

From the standpoint of the dynamic semantics, this extension requires a reformulation of the evaluation rules defined in the previous chapter 2, in order to incorporate the notions of interleaving and events. From the standpoint of typing, it can be straightforwardly integrated in our system.

Section 2.10.1 addresses related work and presents syntactic extensions of our language, and then the new objects and rules of its dynamic semantics. The extension of the static semantics is given section 2.10.2 and its consistency with the dynamic semantics is stated section 2.10.3. Section 2.10.4 gives some examples of use.

2.10.1 Dynamic Semantics

Various calculus have been proposed for reasoning about communicating processes: the communicating sequential processes of [Hoare, 1985], the calculus of communicating systems of [Milner, 1990] and more recently the pi-calculus [Milner & al., 1992]. The one presented in this section is strongly related to the proposals of [Leroy, 1992].

Communications and concurrency are provided by new operators and language constructs. The procedure `channel` allocates a communication channel. A value can be sent to by the operation `to` and received from by using `from`. The new language constructs are the form `either` which non-deterministically evaluates `e` or `e'` and, in contrast, the form `(cobegin e e')` that evaluates `e` or `e'` concurrently.

`op`	::=	`open` |	allocation
		`to` |	emission
		`from`	reception
`e`	::=	`(either e e')` |	non-deterministic choice
		`(cobegin e e')`	parallel evaluation

Extended Syntax

These new operations correspond new semantic objects. Communication channels p are first class objects involved in communication events, of the form $(p?v)$ and $(p!v)$, which denote a reception request and an emission event respectively. Sequences of events form message queues m. The symbol ϵ traditionally denotes a void message queue. Traces f are adapted to register communication events during the evaluation of programs.

v	$\in$	$Value$	$=\{u\}+Port+Closure$	values
p	$\in$	$Port=Ref$		channel
e, $(p?v)$, $(p!v)$	$\in$	$Event$		events
m, ϵ	$\in$	$Message$	$=Event^*$	queue
f	$\in$	$Trace$	$=\mathcal{P}(open(Port)+in(Port)+out(Port))$	trace of events

Channels, Events and Message Queues

Communication channels p are, just like locations, defined on a countable set of symbolic objects and are, for convenience, represented on the same semantic domain as locations. As in the previous section, the evaluation of expression is presented by a "big step" semantics, which relates expressions `e` to values v in an environment E. In contrast to the previous section, however, we do not need to describe a store. We collect communication events m and a trace f.

$$\frac{}{E \vdash \mathtt{x} \rightarrow E(\mathtt{x}), \epsilon, \emptyset} \quad \text{(var)}$$

$$\frac{}{E \vdash \mathtt{(lambda\ (x)\ e)} \rightarrow (\mathtt{x}, \mathtt{e}, E_{\mathtt{x}}), \epsilon, \emptyset} \quad \text{(abs)}$$

$$\frac{E \vdash \mathtt{e} \rightarrow v, m, f \qquad E_{\mathtt{x}} + \{\mathtt{x} \mapsto v\} \vdash \mathtt{e'} \rightarrow v', m', f'}{E \vdash \mathtt{(let\ (x\ e)\ e')} \rightarrow v', m.m', f \cup f'} \quad \text{(let)}$$

$$\frac{E \vdash \mathtt{e} \rightarrow (\mathtt{x}, \mathtt{e''}, E'), m, f \qquad E \vdash \mathtt{e'} \rightarrow v', m', f' \qquad E' + \{\mathtt{x} \mapsto v'\} \vdash \mathtt{e''} \rightarrow v'', m'', f''}{E \vdash \mathtt{(e\ e')} \rightarrow v'', m.m'.m'', f \cup f' \cup f''} \quad \text{(app)}$$

Dynamic Semantics

The rules above mainly read like those of the previous section. The rule (either) describes the non-deterministic choice of evaluating `e` or `e'` by the construct `either` using two rules. The other rule, (cobegin), is more involved and introduces parallelism. Its expression using a big-step semantics is adapted from [Leroy, 1992]. It separates the concurrent evaluation of two subexpressions `e` and `e'` from the resolution of interleavings between communication events.

$$\frac{E \vdash \mathtt{e} \rightarrow v, m, f}{E \vdash \mathtt{(either\ e\ e')} \rightarrow v, m, f} \quad (\text{either})_1$$

$$\frac{E \vdash \mathtt{e'} \rightarrow v, m, f}{E \vdash \mathtt{(either\ e\ e')} \rightarrow v, m, f} \quad (\text{either})_2$$

$$\frac{E \vdash \mathtt{e} \rightarrow v, m, f \qquad E \vdash \mathtt{e'} \rightarrow v', m', f' \qquad m \parallel m' \triangleright m''}{E \vdash \mathtt{(cobegin\ e\ e')} \rightarrow (v, v'), m'', f \cup f'} \quad (\text{cobegin})$$

Dynamic Semantics of Concurrency

Communication events are reduced according to the following rules that express all possible interleavings which may occur during the concurrent reduction of communication events. The first rule reads as the termination of communications: the reduction of two message queues leads to the empty message.

The second rule is the commutativity rule of the operator $\parallel$ and allows to choose different reduction strategies. The third rule tells how the reduction a message queue can be propagated. The last rule tells how communication events are reduced: $(p!v)$ communicates the value v along the channel p to the answer the request $(p?v)$.

$$\frac{}{\epsilon \parallel \epsilon \triangleright \epsilon} \qquad \frac{m \parallel m' \triangleright m''}{m' \parallel m \triangleright m''} \qquad \frac{m \parallel m' \triangleright m''}{m \parallel m'.e \triangleright m''.e} \qquad \frac{m \parallel m' \triangleright m''}{m.(p!v) \parallel m'.(p?v) \triangleright m''}$$

Dynamic Semantics for "Rendez-Vous"

We shall now show how such message queues are built by giving the dynamic semantics for communication operations. First, the operation `open` opens a new communication channel p which the effect $open(p)$ traced. Second, the operation `from` requests a value v on the channel p. Finally, the form `to` sends a value v along on the communication channel p.

$$\frac{p \text{ is fresh}}{E \vdash \mathtt{(open)} \rightarrow p, \epsilon, \{open(p)\}} \quad (\text{open})$$

$$\frac{E \vdash \mathtt{e} \rightarrow p, m, f}{E \vdash \mathtt{(from\ e)} \rightarrow v, m.(p?v), f \cup \{in(p)\}} \quad (\text{in})$$

$$\frac{E \vdash \mathtt{e} \rightarrow p, m, f \qquad E \vdash \mathtt{e'} \rightarrow v, m', f'}{E \vdash \mathtt{((to\ e)\ e')} \rightarrow u, m.m'.(p!v), f \cup f' \cup \{out(p)\}} \quad (\text{out})$$

Dynamic Semantics of Communication

Communication traces are constructed and collected in a similar fashion than in the previous chapter. Each of the three communication operations `open`, `from` and `to` introduced a different dynamic effect $open(p)$, $in(p)$ or $out(p)$ in the trace of communications.

2.10.2 Static Semantics

Adapting the static semantics to incorporate communications and concurrency appears to be dramatically simple. It only requires the semantic domains of types to be extended to understand the type of pairs and communication channels, and the domain of effects to understand communication effects $open(\rho)$ for the communication channel initialization effect, $in(\rho)$, for the reception effect and $out(\rho)$ for the emission effect.

$$\begin{array}{lll} \sigma & ::= \ldots \mid open(\rho) \mid in(\rho) \mid out(\rho) & \text{communication effects} \\ \tau & ::= \ldots \mid \tau \times \tau \mid chan_\rho(\tau) & \text{pair and channel types} \end{array}$$

Static Semantics Objects

The static semantics for the new language constructs implementing concurrency is simple. It gathers the type and effect τ, τ' and σ, σ' collected in the subexpression `e` and `e'` of the constructs `either` and `cobegin`.

$$\frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \qquad \mathcal{E} \vdash \mathtt{e'} : \tau, \sigma'}{\mathcal{E} \vdash \mathtt{(either\ e\ e')} : \tau, \sigma \cup \sigma'} \quad \text{(either)}$$

$$\frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \qquad \mathcal{E} \vdash \mathtt{e'} : \tau', \sigma'}{\mathcal{E} \vdash \mathtt{(cobegin\ e\ e')} : \tau \times \tau', \sigma \cup \sigma'} \quad \text{(cobegin)}$$

Static Semantics for Concurrency

The static semantics of communication operations describes, by using axioms and rules, the introduction of channel types $chan_\rho(\tau)$ and communication effects, $open(\rho)$, $in(\rho)$ and $out(\rho)$ by the different primitives.

$$\mathcal{E} \vdash \mathtt{(open)} : chan_\rho(\tau), open(\rho) \quad \text{(channel)}$$

$$\mathcal{E} \vdash \mathtt{from} : chan_\rho(\tau) \stackrel{\sigma \cup in(\rho)}{\rightarrow} \tau, \emptyset \quad \text{(from)}$$

$$\mathcal{E} \vdash \mathtt{to} : chan_\rho(\tau) \stackrel{\sigma}{\rightarrow} \tau \stackrel{\sigma' \cup out(\rho)}{\rightarrow} unit, \emptyset \quad \text{(to)}$$

Static Semantics for Communication

2.10.3 Consistency

The consistency of the extension of our language to communications and concurrency is established by adapting the relations presented in the previous section. This is, fortunately, not of any particular difficulty.

Definition 2.7 (Consistent trace and effect) *Let $\mathcal{S}$ a model, f a communication trace and σ an effect.*

$$\mathcal{S} \models f : \sigma \Leftrightarrow \begin{cases} \forall\ open(p) \in f, & \mathcal{S}(p) = (\rho, \tau) \ \wedge \ open(\rho) \in \sigma \\ \forall\ in(p) \in f, & \mathcal{S}(p) = (\rho, \tau) \ \wedge \ in(\rho) \in \sigma \\ \forall\ out(p) \in f, & \mathcal{S}(p) = (\rho, \tau) \ \wedge \ out(\rho) \in \sigma \end{cases}$$

Lemma 2.11 (Traces and Models Extension) *If $\mathcal{S} \sqsubseteq \mathcal{S}'$ and $\mathcal{S} \models f : \sigma$ then $\mathcal{S}' \models f : \sigma$. Also if $\mathcal{S} \models f : \sigma$ then $\mathcal{S} \models f : \sigma \cup \sigma'$ and $\theta\mathcal{S} \models f : \theta\sigma$ for any θ.*

Proof Same as for the lemmas 2.5 and 2.6 □

Definition 2.8 (Consistent Events) *A message queue m is consistent with the model $\mathcal{S}$, written $\models m : \mathcal{S}$, if and only if $Dom(\mathcal{S}) = Dom(m)$ and $\mathcal{S} \models v : \tau$ for every $(p?v)$, $(p!v)$ in m and with $\mathcal{S}(p) = (\rho, \tau)$.*

Lemma 2.12 (Events and Semantic Substitution) *If $\models m : \mathcal{S}$ then $\models m : \theta\mathcal{S}$ for any θ.*

Proof Same as for the lemma 2.7 □

Theorem 2.4 (Consistency of dynamic and static semantics) *If* $\mathcal{S} \models E : \mathcal{E}$, $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ *and* $E \vdash \mathtt{e} \rightarrow v, m, f$ *then there exists* $\mathcal{S}'$ *extending* $\mathcal{S}$ *such that* $\mathcal{S}' \models v : \tau$, $\mathcal{S}' \models f : \sigma$ *and* $\models m : \mathcal{S}'$.

Proof The proof is by induction on derivation. It only differs from the proof of theorem 2.1 by the concurrency constructs. The case analysis of the communication operations (channel), (in) and (out) are equivalent to the case analysis of (new), (get) and (set) developed in theorem 2.1 and does not need to be repeated.

Case of (either) By hypothesis, $\mathcal{S} \models E : \mathcal{E}$, $\mathcal{E} \vdash$ `(either e e')` $: \tau, \sigma \cup \sigma'$ and $E \vdash$ `(either e e')` $\rightarrow v, m, f$. By definition of the rule (either) in the static semantics, $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ and $\mathcal{E} \vdash \mathtt{e}' : \tau, \sigma'$. By definition of (either) in the dynamic semantics, we have either $E \vdash \mathtt{e} \rightarrow v, m, f$ or $E \vdash \mathtt{e}' \rightarrow v, m, f$. By induction hypothesis on `e`, there exists $\mathcal{S}'$ extending $\mathcal{S}$ such that $\mathcal{S}' \models v : \tau$, $\mathcal{S}' \models f : \sigma$ and $\models m : \mathcal{S}'$. Same for `e'`.

Case of (cobegin) By hypothesis, $\mathcal{E} \vdash$ `(cobegin e e')` $: \tau \times \tau', \sigma \cup \sigma'$ and $E \vdash$ `(cobegin e e')` $\rightarrow v, m'', f \cup f'$. By definition of the rule (cobegin) in the static semantics, $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ and $\mathcal{E} \vdash \mathtt{e}' : \tau', \sigma'$. By definition of (cobegin) in the dynamic semantics, we have $E \vdash \mathtt{e} \rightarrow v, m, f$, $E \vdash \mathtt{e}' \rightarrow v', m', f'$ and $m \parallel m' \triangleright m''$.

By induction hypothesis on `e`, there exists $\mathcal{S}_1$ extending $\mathcal{S}$ such that $\mathcal{S}_1 \models v : \tau$, $\mathcal{S}_1 \models f : \sigma$ and $\models m : \mathcal{S}_1$. By the lemma 2.3, $\mathcal{S}_1 \models E : \mathcal{E}$. By induction hypothesis on `e'`, there exists $\mathcal{S}'$ extending $\mathcal{S}_1$ such that $\mathcal{S}' \models v' : \tau'$, $\mathcal{S}' \models f' : \sigma'$ and $\models m' : \mathcal{S}'$. By the lemma 2.3, $\mathcal{S}' \models v : \tau$. By the lemma 2.11, $\mathcal{S}' \models f : \sigma$. By the lemma 2.12, $\models m : \mathcal{S}'$.

From $\models m : \mathcal{S}'$ and $\models m' : \mathcal{S}'$, we wish to show that $\models m'' : \mathcal{S}'$ where $m \parallel m' \triangleright m''$. This is done by induction on the reduction. First, if $m = m' = \epsilon$ then $\models m'' : \mathcal{S}'$ by definition of $\models$.

Second, suppose that $m_1.e \parallel m' \triangleright m_1''.e$ (or $m' \parallel m_1.e \triangleright m_1''.e$, respectively) where $m = m_1.e$ and $m'' = m_1''.e$. By definition of $\parallel$, this requires that $m_1 \parallel m' \triangleright m_1''$. By induction hypothesis on the reduction of m_1 with m', $\models m_1'' : \mathcal{S}'$. Since $\models m : \mathcal{S}'$, $\models e : \mathcal{S}'$. Thus, $\models m'' : \mathcal{S}'$.

Finally, suppose that $m_1.(p?v) \parallel m_1'.(p!v) \triangleright m''$ (or $m.(p?v) \parallel m'.(p!v) \triangleright m''$, respectively) where $m = m_1.(p?v)$ and $m' = m_1'.(p!v)$. This requires that $m_1 \parallel m_1' \triangleright m''$. By induction hypothesis on the reduction of m_1 and m_1', we get that $\models m'' : \mathcal{S}'$ and conclude □

2.10.4 Example

Before concluding, we describe a classical programming exercise. The function `stamp` emits a sequence of integers over a channel given as argument. This emission is reseted to zero when the process receives a message on the other channel. Such a function is very useful in practice for serving unique stamps to different concurrent processes.

```
(define stamp (lambda (output reset)
                ((rec (f n)
                  (either (begin ((to output) n) (f (+ n 1)))
                          (begin (from reset) (f 0))))
                 0)))
```

Our type and effect system will document the function `stamp` by giving the following information.

$$\mathtt{stamp} : chan_{\rho}(int) \times chan_{\rho'}(\tau) \xrightarrow{\sigma' \cup out(\rho) \cup in(\rho')} \tau' \blacksquare$$

We have presented an extension of our type and effect system to incorporate the typing of communication operations and concurrency language constructs. It shows up to be simple to integrate to our static semantics as well in the dynamic semantics, thanks to its "big-step" formulation.

2.11 Related Work

In the following, we give a description of the related work on alias and effect analysis. It mainly refers to [Deutsch, 1992], chapter 4.1, which gives a complete and insightful classification of the very wide literature on the subject.

It appears that there are a very few alias and effect analysis that address full-fledged functional programming languages, without limitations to first order functions or on simple data structures. There are, surprisingly, even a fewer of such analysis that have been proved correct.

Abstract interpretation [Cousot & Cousot, 1977] is the most popular method for the analysis of alias and effects in functional languages. The related work often uses complex representations of abstracted locations, stores and states via graphs [Deutsch, 1990, Harrison, 1989, Larus & Hilfinger, 1988, Neirynck & al., 1989, Shivers, 1991, Stransky, 1988].

To deal with higher-order functions, analysis of alias usually requires a global (interprocedural) control flow analysis, that finitely partitions a given program into a set of function invocation contexts and then to re-analize the functions in every such context of the program [Harrison, 1989, Neirynck & al., 1989, Shivers, 1991, Stransky, 1988].

Such a mechanism incurs a very heavy computational cost [Rosen, 1979]. However, some of these analysis are capable of saving part of the properties of the analyzed functions, which are instantiated when re-analized, thus saving a valuable amount of time [Deutsch, 1990, Larus & Hilfinger, 1988].

On the opposite, type and effect inference is a typical example of an on-line analysis, in that it associates a simple and polymorphic representation of alias and effect information with every function in a program.

The advantage of an on-line analysis is, in general, that it can be used in a programming environment that support separate compilation mechanisms, while a global analysis cannot.

Another approach used to determine alias in functional programs is linear logic [Girard, 1987]. Analysis techniques based on linear logic address a simpler problem than determining alias relations, the problem of single threadedness [Chirimar & al., 1992, Guzman & Hudak, 1990, Odersky, 1992, Wadler, 1991]. It aims at answering the question "Is this datum aliased to any other?". By representing uniform alias relations using regions, type and effect inference certainly provides richer information than this approach does.

Using the framework of abstract interpretation, another approach to alias analysis is the determination of *non-uniform* alias relations, either by global or on-line analysis [Hendren, 1990, Deutsch, April 1992]. Unlike a *uniform* alias analysis which typically tells that "The `cars` of list l_1 are shared with the `cars` of list l_2" (Type and effect system belongs to this category "The car of list l_1 and l_2 are in the same region"), one may detect, with a non uniform alias analysis, that "All $2 * n$ `cars` of list l_1 are shared to all $2 * m + 1$ `cars` of list l_2".

Non uniform aliasing analysis seem interesting for doing program verifications such as the checking of non-interference in parallel languages or the checking of safe explicit deallocation. They use however very complex state representations and are for the time being limited to the analysis of first-order functions.

Being more general and less complex, uniform aliasing analysis, such as type and effect inference, seems on the other hand worthy for the integration into realistic compilers for functional languages and for a simple formalization of related program optimizations.

2.12 Conclusion

We have presented an inference system which reconstruct the principal type, region and effect of expressions for a polymorphic functional language extended with imperative constructs. This system extends the principle of polymorphic type inference of [Milner, 1978] to the reconstruction of regions and effects by the introduction of a tantamount notion to subtyping in the domain of effects: *subeffecting*.

We have proved the consistency of our inference system and presented a correct algorithm which computes the principal type, region and effect of expressions with respect to substitution on variables and the minimal effect with respect to the rule of subsumption on effects.

A number of standard program optimizations can take advantage of the program properties that type and effect inference computes. This framework provides the basis for sophisticated program verification and

transformation techniques in the presence of side-effects and higher-order functions. Sophisticated compiler optimizations such as stack allocation and parallel code generation have been discussed in this chapter.

Applications are more extensively developed in the chapter 4. The type and effect of expressions, annotated with regions, are of a suitable generality and a sufficient precision to fulfill present needs for analysis techniques for the efficient implementation of functional languages.

Type and effect inference, as a semantic analysis technique, establishes a very strong relation between structural information, *types*, relational information, *regions* and behavioral information, *effects*. This relation eases the specification of optimization techniques based on semantic information; it use proofs techniques based on the lambda-calculus and offers precise and comprehensible documentation about programs.

Chapter 3

The Type and Effect Discipline

Static typing is the most widely used technique of static analysis in programming languages. The strength of static typing is that the successful type checking of a program guarantees the absence of type errors in it. However, because typing rules must be simple and because programs can be complex, type systems often reject programs which are correct but cannot be recognized as such. This tradeoff, between simplicity and effectiveness, has motivated the introduction. Type polymorphism is powerful because it statically types generic functions. Generic functions can operate on data of different structures.

Example Consider the following. The higher order function `map` below is a typical example of generic function. It takes two arguments: a function `f` and a list `l`. The function `null?` first tests if the list `l` is empty. In this case, it returns the empty list `nil`. Otherwise, it builds a pair, using `cons`, made of the result of the call to `f` with the first element of `l`, the `car` of `l`, and the result of recursively calling `map` with `f` and the rest of the list `l`, its `cdr`.

```
(define map (lambda (f l)
              (if (null? l) nil
                  (cons (f (car l)) (map f (cdr l))))))
```

By looking at the definition of `map`, it is evident that the elements of the list `l` and `f`'s formal argument must have matching types (mapping the boolean function `not` over a list of integers would cause a type error). Nonetheless, it is clear that the function `map` is implicitly defined regardless of the type of these elements ■

Generic functions, such as the function `map`, are interesting in that they can be reused in many programs without modification. In practice, many other list processing functions can be defined regardless of the element's type of the list they manipulate: reversing functions, sorting functions or scanning functions. This holds for other data structures as well: hash tables, trees and graphs.

Milner's type system expresses polymorphism in `let` syntactic constructs. Understanding the `let` as an abbreviation offers a simple explanation of polymorphism. Semantically, the expression `(let (x e) e′)` has the same meaning as `e′[e/x]`, the capture-avoiding substitution of `e` for the free occurrences of `x` in `e′`. In typing the substituted expression `e′[e/x]`, each occurrence of the bound expression `e` may have a different type. In Milner's typing discipline. This is expressed by a type scheme which is associated with `x` and represents the possible types of `e`.

Example In the following example, `x` is `id` and `e` is `(lambda (x) x)`.

```
(let (id (lambda (x) x))
  (id 1)
  (id true)))
```

The first occurrence of `id` must be typed as a function from integers to integers and the other as a function from booleans to booleans ■

To type a `let` expression, we associate with `x` all possible types of `e`. Each occurrence of `x` in the body `e'` may have any of these types. In the previous example, `id` is associated with the set of types $\{\tau \xrightarrow{\sigma} \tau \mid \tau \in \mathit{Type},\ \sigma \in \mathit{Effect}\}$ which is represented by the polymorphic type $\forall\alpha\varsigma.\alpha \xrightarrow{\varsigma} \alpha$

Type polymorphic is appropriate for functional programming languages. But the imperative style of programming is usually defined as opposed to functional programming. It is usually associated with the use of operations that permit in-place modification of mutable data structures, such as pointers or communication channels. When adding imperative features to the language, it becomes necessary to introduce a notion of state to understand the meaning of programs. This addition suffices to invalidate our previous explanation of `let`-expressions as abbreviations: the expression `(let (x e) e')` no longer has the same meaning as `e'[e/x]`.

To type pointers in ML, one can think of introducing the type $\mathit{ref}(\tau)$ to represent pointers referencing a value of type τ. Then, one can of typing the pointer initialization procedure `new` by $\tau \rightarrow \mathit{ref}(\tau)$, because it returns a pointer initialized to the given argument, which can have any type τ. However, once initialized, that pointer must always be associated with the same type. Otherwise, one could initialize it with an *int* value, then read it and claim it has type *bool*.

Example The following example illustrates that the naïve extension of Milner's typing discipline to operations on pointers is unsound in the presence of polymorphism.

```
(let (rid (new (lambda (x) x)))
  (set rid (lambda (y) (+ y 1)))
  ((get rid) true)))
```

In ML, the most general type of the expression `(new (lambda (x) x))` is $\mathit{ref}(\alpha \rightarrow \alpha)$. Generalizing the free type variable α yields the type scheme $\forall\alpha.\mathit{ref}(\alpha \rightarrow \alpha)$ of `rid`. It can be instantiated to $\mathit{ref}(\mathit{int} \rightarrow \mathit{int})$ to type the assignment operation `(set rid (lambda (y) (+ y 1)))` and then to $\mathit{ref}(\mathit{bool} \rightarrow \mathit{bool})$ to type the dereference operation `((get rid) true)`. However, trying to evaluate the program above leads to a type error, by attempting to add `1` to `true` ■

This example shows that static typing must restrict the use of polymorphism over mutable values. Just as references change the semantics of `let` expressions, they also necessitate a change in how `let` expressions are typed: an accessible reference, once created, shall always be used with the same type. To implement such a limitation, several type systems have been proposed that consist of restricting the type generalization of expressions bound by `let` constructs [Leroy, 1990, Milner & al., 1990, Tofte, 1987, Wright, 1992]. All these approaches build conservative approximations of types that may be accessible from the global store and turn out to be restrictive in practice by prohibiting generic functions that create temporary mutable structures or by being non-conservative over ML.

In contract to these approach, our typing discipline uses effect inference to approximate the store transformation that is performed when allocating references. It infers initialization effects, of the form $\mathit{init}(\rho, \tau)$, which tell the type τ of data referenced by pointers of the region ρ. This information is used at `let` boundaries to prohibit type generalization over the type τ of referenced values. In order to limit the reporting of such effects to those that affect accessible values, we use an observation criterion which precisely delimit the lexical scope of operations on pointer regions. This is a crucial aspect, as we want to distinguish the function which uses temporary references to implement their purely functional counterpart with an imperative style.

Example The following implementation of the function `map` illustrates this imperative programming style. It uses two temporary references `r` and `x` which point to the list of unprocessed elements and to the intermediate results.

```
(define map (lambda (f l)
              (let ((r (new nil))(x (new l)))
                (until (null? (get x))
                  ((set r) (cons (f (car (get x))) (get r)))
                  ((set x) (cdr (get x))))
                (reverse (get r)))))
```

When all elements of the list `l` are processed, the list pointed at by `r` is reversed by `reverse`, yielding the result. Because the use of the references `r` and `x` is temporary, the function `map` has the same typing constraints as its purely functional implementation and is generic over its arguments ■

Plan

In this chapter, section 3.1 presents the static semantics of the language. The reconstruction algorithm is presented in section 3.4. We state that the static and dynamic semantics are consistent section 3.3 and that our algorithm is correct with respect to the static semantics section 3.6. Before concluding in 3.11, we give a detailed comparison of our type system with the related work (section 3.8) in section 3.9, showing that our approach surpasses many aspects of previous techniques. The technical results presented in this chapter are inspired from [Talpin & Jouvelot, June 1992].

3.1 Static Semantics

The presentation of our static semantics begins with the definition of our new algebra of effects, type schemes and environments. Then we present the rules of the static semantics and discuss about defining an observation criterion for effects. Finally we state the most important formal properties of our static semantics.

3.1.1 Semantics Objects

In contrast to the previous chapter, we consider typed store effects. These effects σ can either be the constant $\emptyset$, that represents the absence of effects, effect variables ς, or store effects $init(\rho,\tau)$, $read(\rho,\tau)$ and $write(\rho,\tau)$, that approximate memory side-effects on the region ρ of references to values of type τ. The range of an effect σ, written $Rng(\sigma)$, is the set of pairs (ρ,τ) such that either $init(\rho,\tau)$, $read(\rho,\tau)$ or $write(\rho,\tau)$ is in σ. We write $Regs(\sigma)$ the set of regions ρ such that (ρ,τ) is in the range of σ.

$$
\begin{array}{rcll}
\sigma & ::= & init(\rho,\tau) \mid read(\rho,\tau) \mid write(\rho,\tau) \mid \emptyset \mid \varsigma \mid \sigma \cup \sigma & \text{effects} \\
\forall\vec{v}.\tau & \in & \text{TyScheme} & \text{type schemes} \\
\mathcal{E} & \in & \text{TyEnv} & \text{type environments}
\end{array}
$$

Typed Effects, Type Schemes and Environments

We use type schemes, introduced by [Milner, 1978], to generically represent the different types of an expression. A type scheme $\forall\vec{v}.\tau$ consists of a types τ which is universally quantified over sequences $\vec{v}$ of type variables, region variables and effect variables. A type τ' is an instance of $\forall\vec{v}.\tau$, written $\tau' \preceq \forall\vec{v}.\tau$, if the variables $\vec{v}$ can be substituted by θ so that $\tau' = \theta\tau$.

In the sequence $\vec{v}$, the variables are assumed to be distinct and their order of occurrence is not significant. When that sequence is empty, we do not distinguish τ from $\forall.\tau$. We identify type schemes that differ only by a renaming of their quantified variables or that differ by the introduction or elimination of quantified variables that are not free in the body of the type scheme.

The context in which an expression is associated with a type and an effect is represented by a type environment $\mathcal{E}$ which maps value identifiers to type schemes. The definitions of free variables and free regions are extended to type schemes by $fv(\forall\vec{v}.\tau) = fv(\tau) \setminus \vec{v}$ and to environments $\mathcal{E}$ by considering that a type variable is free in $\mathcal{E}$ if and only if it is free in $\mathcal{E}(\mathtt{x})$ for some value identifier $\mathtt{x}$ in $Dom(\mathcal{E})$. Substitutions are also extended to type schemes $\forall\vec{v}.\tau$ by using alpha-renaming of quantified variables in type schemes to avoid capture of bound variables.

We extend substitutions to type schemes $\theta(\forall\vec{v}.\tau) = \forall\vec{v'}.\theta''(\theta'\tau)$ modulo renaming of bound variables $\vec{v}$ by fresh $\vec{v'}$ e.g. the sequence of variables $\vec{v'}$ is not free in $\theta\tau$, $\theta' = \{\vec{v} \mapsto \vec{v'}\}$. θ'' is defined by v if $v \in \vec{v'}$ and θv otherwise. The image $\theta(\mathcal{E})$ of a type environment by a substitution is also defined straightforwardly by $(\theta\mathcal{E})(\mathtt{x}) = \theta(\mathcal{E}(\mathtt{x}))$ for every $\mathtt{x}$ in $Dom(\mathcal{E})$. Our extension of substitution on type schemes and environments supports the following lemma.

Lemma 3.1 (Substitution and Instantiation) *If $\tau_1 \preceq \theta(\forall \vec{v}.\tau)$ then there exist θ_2 and $\tau_2 \preceq \forall \vec{v}.\tau$ such that $\tau_1 = \theta_2\tau_2$.*

Proof By hypothesis, $\tau_1 \preceq \theta(\forall \vec{v}.\tau)$. By definition, $\theta(\forall \vec{v}.\tau) = \forall \vec{v'}.(\theta''(\theta'\tau))$ where $\vec{v'}$ is not free in $\theta\tau$, $\theta' = \{\vec{v} \mapsto \vec{v'}\}$ and θ'' is defined by v if $v \in \vec{v'}$ and θv otherwise. By definition of $\preceq$, there exists θ'_1 defined on $\vec{v'}$ such that $\tau_1 = \theta'_1(\theta''(\theta'\tau))$. Let $\theta'_2 = \{\vec{v} \mapsto \theta'_1\vec{v'}\}$ then $\tau_2 = \theta'_2\tau \preceq \forall \vec{v}.\tau$ and the restriction θ_2 of θ on $fv(\tau_2) \setminus \vec{v}$ verifies that $\tau_1 = \theta_2\tau_2$ □

3.1.2 Type Generalization

The generalization $Gen(\mathcal{E}, \sigma)(\tau)$ of a type τ is performed at `let` boundaries on some of the type, region and effect variables $\vec{v}$ that occur free in τ. A variable cannot be generalized when it is either free in the type environment $\mathcal{E}$ or present in the observed effect σ.

$$Gen(\mathcal{E}, \sigma)(\tau) = \text{let } \vec{v} = fv(\tau) \setminus (fv(\mathcal{E}) \cup fv(\sigma)) \text{ in } \forall \vec{v}.\tau$$

Type Generalization

The first condition is standard in purely functional languages. As for the second, just as types are bound to identifiers in the environment, types are bound to regions in the reconstructed allocation effects. Thus, when these regions are observable from the context i.e. in the type environment $\mathcal{E}$ or the type τ of the returned value, those types cannot be generalized.

3.1.3 Rules of the Static Semantics

The next figure summarizes the rules of our new static semantics. The rules (abs) and (app) are the same as in the previous chapter. The axioms associated with the store operations `new`, `get` and `set` are also the same.

$$\frac{\tau \preceq \mathcal{E}(\mathtt{x})}{\mathcal{E} \vdash \mathtt{x} : \tau, \emptyset} \quad \text{(var)}$$

$$\frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \qquad \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\sigma, \mathcal{E})(\tau)\} \vdash \mathtt{e'} : \tau', \sigma'}{\mathcal{E} \vdash \mathtt{(let\ (x\ e)\ e')} : \tau', \sigma \cup \sigma'} \quad \text{(let)}$$

$$\frac{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma}{\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\sigma} \tau', \emptyset} \quad \text{(abs)}$$

$$\frac{\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\sigma} \tau', \sigma' \qquad \mathcal{E} \vdash \mathtt{e'} : \tau, \sigma''}{\mathcal{E} \vdash \mathtt{(e\ e')} : \tau', \sigma \cup \sigma' \cup \sigma''} \quad \text{(app)}$$

Static Semantics

Our new static semantics manipulates type schemes by using the type generalization/instantiation mechanism specified in the previous section. For type generalization, only side-effects that can affect the context of an expression are worth reporting. The other effects of σ are not observable from the enclosing context. They do not have to be reported since they are only local.

3.1.4 Observation Criterion

The effect of a function derives statically from the state transforming operations that the expression it abstracts performs when it is executed. For example, store operations comprise the initialization, reading and writing of references and are approximated on regions.

Even if a value expression performs certain operations on the store, one may be able to detect that those operations cannot interfere with other expressions. This is the case when the regions over which the effect ranges are unreferenced in the rest of the program. If this is the case, then we shall mask effects which derive from those operations.

Example Consider the following derivation. We examine a function that initializes and dereferences a temporary pointer that successively points to the argument and the result.

$$\dfrac{\dfrac{\dfrac{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{x} : \tau, \emptyset}{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{(new\ x)} : \mathit{ref}_{\rho}(\tau), \mathit{init}(\rho, \tau)}}{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{(get\ (new\ x))} : \tau, \mathit{init}(\rho, \tau) \cup \mathit{read}(\rho, \tau)}}{\mathcal{E} \vdash \mathtt{(lambda\ (x)\ (get\ (new\ x)))} : \tau \xrightarrow{\mathit{init}(\rho,\tau) \cup \mathit{read}(\rho,\tau)} \tau, \emptyset}$$

The expression `(get (new x))` has an initialization effect $\mathit{init}(\rho, \tau)$ and a read effect $\mathit{read}(\rho, \tau)$ on a certain region ρ. The derivation does not constrain the choice of the region ρ. Moreover, this region does not refer to any other reference in the scope of the expression: it is neither related to $\mathcal{E}$ nor to τ.

$$\dfrac{\ldots\ \rho \notin \mathit{fr}(\mathcal{E}) \cup \mathit{fr}(\tau)\ \ldots}{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{(get\ (new\ x))} : \tau, \emptyset}$$

We shall thus be able to detect that it is local to the body of the function and avoid the burden of reporting the initialization and read effects related to it outside of the scope of its body. This could be performed by the informal derivation above ■

In this chapter, we adapt of our type and effect inference system so as to statically determine the lexical scope of data regions by using an observation criterion. The observation of effects consists of selecting one of them which ranges over regions that refer to data accessible in the environment of an expression or in its value. The accessible data is abstracted by the free regions of the type environment and the value type of the expression.

In the static semantics, only side-effects that can affect the typing context of an expression, i.e. its type environment $\mathcal{E}$ and its value type τ, are worth reporting. The other effects of σ refer to local references that are freshly created and not exported from the expression `e`.

$$\dfrac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma \qquad \sigma' \supseteq \mathit{Observe}(\mathcal{E}, \tau)(\sigma)}{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'} \quad \text{(sub)}$$

Observation and Subsumption Rule

The observation criterion is specified by the rule (sub) which tells that an expression `e` has any effect σ' bigger than the observable effects that can be inferred for it. $\mathit{Observe}(\mathcal{E}, \tau)(\sigma)$ is the set of observable effects of σ.

$$\mathit{Observe}(\mathcal{E}, \tau)(\sigma) = \{\mathit{init}(\rho, \tau'), \mathit{read}(\rho, \tau'), \mathit{write}(\rho, \tau') \in \sigma \mid \rho \in \mathit{fr}(\mathcal{E}) \cup \mathit{fr}(\tau)\} \cup \{\varsigma \in \sigma \mid \varsigma \in \mathit{fv}(\mathcal{E}) \cup \mathit{fv}(\tau)\}$$

Observable Effects

The observable effects are the effect variables ς that occur free in τ or $\mathcal{E}$ and the effects of the form $init(\rho,\tau')$, $read(\rho,\tau')$ and $write(\rho,\tau')$ where ρ occurs free in τ or in $\mathcal{E}$. We write $Observe(\tau)(\sigma)$ for $Observe(\{\},\tau)(\sigma)$ and $Observe(\sigma)(\sigma')$ for $Observe(unit \xrightarrow{\sigma} unit)(\sigma')$. The functions $Observe$ has the following formal properties which are widely used in the rest of this chapter.

Lemma 3.2 (*Observe* and free variables) *If $\sigma' = Observe(\tau)(\sigma)$ then $fr(\tau) \cap Regs(\sigma \setminus \sigma') = \emptyset$ and $fr(\sigma') \cap Regs(\sigma \setminus \sigma') = \emptyset$.*

Proof By hypothesis, $\sigma' = Observe(\tau)(\sigma)$. By definition of $Observe$ and for any $\rho \in Regs(\sigma \setminus \sigma')$, we have that $\rho \notin fr(\tau) \cup fr(\sigma')$. Thus, $(fr(\tau) \cup fr(\sigma')) \cap Regs(\sigma \setminus \sigma') = \emptyset$ as expected □

By the lemma 3.3, we show that the observation is stable under substitution, in that combinations of substitutions θ to the function $Observe$ can be compared with the application of the function $Observe$to substituted terms.

Lemma 3.3 (*Observe* and substitution) *If $\sigma' = Observe(\tau)(\sigma)$ then $\theta\sigma' \subseteq Observe(\theta\tau)(\theta\sigma)$.*

Proof Let us write $\sigma'' = Observe(\theta\tau)(\theta\sigma)$. We proceed by case analysis. For every $\varsigma \in \sigma'$ and by definition of $Observe$, we have $\varsigma \in fv(\tau)$ and thus $\sigma'' \supseteq \theta\varsigma$. For every $init(\rho,\tau') \in \sigma'$ (respectively, $read(\rho,\tau')$ and $write(\rho,\tau')$) and by definition of $Observe$, we have that $\rho \in fr(\tau)$ and thus $\theta\rho \in fr(\theta\tau)$. This implies that $init(\theta\rho,\theta\tau') \in \sigma''$. This proves that $\sigma'' \supseteq \theta\sigma'$ □

Note, however, that the containment is proper. An example where we do not have $\theta(Observe(\tau)(\sigma)) = Observe(\theta\tau)(\theta\sigma)$ is $\tau = ref_{\rho_1}(int)$, $\sigma = init(\rho_1, int) \cup init(\rho_2, unit)$ and $\theta = \{\rho_2 \mapsto \rho_1\}$.

3.2 Formal Properties of the Static Semantics

The lemma of substitution is used both in the proof of consistency and in the proofs of correctness for the reconstruction algorithm. Since we have added the rule (sub) of observation and changed the rule (let) for `let` expressions, we shall formally confirm that it is verified.

Lemma 3.4 (Substitution) *If $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ then $\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \theta\sigma$ for any substitution θ.*

Proof The proof is by induction on the proof derivation. The case analysis of the rules (var), (abs) and (app) are identical to the proof of lemma 2.1.

Case of (let) By hypothesis $\mathcal{E} \vdash$ `(let (x` $\mathtt{e}_1$`)` $\mathtt{e}_2$`)` $: \tau, \sigma_1 \cup \sigma_2$. By definition of the rule (*let*), we have

$$\mathcal{E} \vdash \mathtt{e}_1 : \tau_1, \sigma_1 \quad \text{and} \quad \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\sigma_1, \mathcal{E})(\tau_1)\} \vdash \mathtt{e}_2 : \tau, \sigma_2$$

By induction hypothesis on e_1, we have $\theta\mathcal{E} \vdash \mathtt{e}_1 : \theta\tau_1, \theta\sigma_1$ for any substitution θ. Let $\forall\vec{v}.\tau_1$ be $Gen(\sigma_1, \mathcal{E})(\tau_1)$. Consider any θ and define θ' as the extension of $\theta_{\vec{v}}$ with $\{\vec{v} \mapsto \vec{v'}\}$, with fresh $\vec{v'}$ and thus not free in $\theta\sigma_1$ or in $\theta\mathcal{E}$. This ensures that $\theta'(\forall\vec{v}.\tau_1) = \forall\vec{v'}.(\theta'\tau_1)$ and:

$$\theta(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\sigma_1, \mathcal{E})(\tau_1)\}) = \theta\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\theta\sigma_1, \theta\mathcal{E})(\theta'\tau_1)\}$$

Using the induction hypothesis on $\mathtt{e}_1$ with θ', we get that $\theta'\mathcal{E} \vdash \mathtt{e}_1 : \theta'\tau_1, \theta'\sigma_1$ and thus , by definition of Gen, $\theta\mathcal{E} \vdash \mathtt{e}_1 : \theta'\tau_1, \theta\sigma_1$. By induction hypothesis on e_2, we get:

$$\theta(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\sigma_1, \mathcal{E})(\tau_1))\} \vdash \mathtt{e}_2 : \theta\tau, \theta\sigma_2$$

which is equivalent to $\theta\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\theta\sigma_1, \theta\mathcal{E})(\theta'\tau_1)\} \vdash \mathtt{e}_2 : \theta\tau, \theta\sigma_2$. By definition of the rule (let), we can then conclude that $\theta\mathcal{E} \vdash$ `(let (x` $\mathtt{e}_1$`)` $\mathtt{e}_2$`)` $: \theta\tau, \theta\sigma$

Case of (sub) By hypothesis, $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$. By definition of the rule (sub), this requires that there exists σ' such that

$$\mathcal{E} \vdash \mathtt{e} : \tau, \sigma' \quad \text{and} \quad \sigma \supseteq \mathit{Observe}(\mathcal{E}, \tau)(\sigma')$$

Let θ be any substitution. By induction hypothesis on the derivation, we have that

$$\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \theta\sigma'$$

Let $\sigma'_1 = \mathit{Observe}(\mathcal{E}, \tau)(\sigma')$ and $\sigma'_2 = \sigma' \setminus \sigma'_1$. Let us define the substitution θ' that maps effect variables in σ'_2 to $\emptyset$ and regions in $\mathit{Regs}(\sigma'_2)$ to fresh regions, not free in $\theta\mathcal{E}$, $\theta\tau$ and $\theta\sigma'_1$. By the lemma 3.2, $\mathit{Regs}(\sigma'_2) \cap (\mathit{fr}(\tau) \cup \mathit{fr}(\mathcal{E}) \cup \mathit{fr}(\sigma'_1)) = \emptyset$. Thus, for any substitution of the form $\theta \circ \theta'$, we have

$$\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \theta(\sigma'_1 \cup \theta'\sigma'_2)$$

By definition of the rule (sub),

$$\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta(\sigma'_1 \cup \theta'\sigma'_2))$$

By definition of *Observe*,

$$\mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta(\theta'\sigma'_2)) = \mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta'\sigma'_2) = \emptyset \quad \text{and} \quad \mathit{Observe}(\theta\sigma'_1)(\theta'\sigma'_2) = \emptyset$$

Thus,

$$\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta\sigma'_1)$$

By definition of *Observe*, $\mathit{Observe}(\theta\mathcal{E}, \theta\tau)(\theta\sigma'_1) \subseteq \theta\sigma'_1$. By the rule (sub), $\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \theta\sigma'_1$. Since $\sigma \supseteq \sigma'_1$, we have $\theta\sigma \supseteq \theta\sigma'_1$, and by the rule (sub),

$$\theta\mathcal{E} \vdash \mathtt{e} : \theta\tau, \theta\sigma \ \square$$

Another significant property of the typing rules, is that if one can prove a type τ and an effect σ for an expression $\mathtt{e}$ with a type environment $\mathcal{E}$, then τ and σ can also be proved for $\mathtt{e}$ with a stronger environment than $\mathcal{E}$.

Lemma 3.5 (Strengthening) *If $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ and if $\tau' \preceq \mathcal{E}(\mathtt{x})$ implies $\tau' \preceq \mathcal{E}'(\mathtt{x})$ for every $\mathtt{x} \in \mathit{Dom}(\mathcal{E})$ then $\mathcal{E}' \vdash \mathtt{e} : \tau, \sigma$.*

Proof The proof is by induction on the derivation. By hypothesis, $\mathcal{E}$ and $\mathcal{E}'$ are such that $\tau'' \preceq \mathcal{E}(\mathtt{x})$ implies that $\tau'' \preceq \mathcal{E}'(\mathtt{x})$ for every $\mathtt{x} \in \mathit{Dom}(\mathcal{E})$.

Case of (var) By hypothesis, $\mathcal{E} \vdash \mathtt{x} : \tau, \emptyset$. By definition of the rule (var), we have that $\tau \preceq \mathcal{E}(\mathtt{x})$. By hypothesis, $\tau \preceq \mathcal{E}'(\mathtt{x})$. By definition of the rule (var), we conclude that

$$\mathcal{E}' \vdash \mathtt{x} : \tau, \emptyset$$

Case of (abs) By hypothesis, $\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\sigma} \tau', \emptyset$. By definition of the rule (abs), $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma$. By induction hypothesis with $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\}$, $\mathcal{E}'_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma$. By definition of the rule (abs),

$$\mathcal{E}' \vdash \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\sigma} \tau', \emptyset$$

Case of (app) By hypothesis, $\mathcal{E} \vdash \mathtt{(e\ e')} : \tau', \sigma \cup \sigma' \cup \sigma''$. By definition of the rule (app) $\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\sigma''} \tau', \sigma$ and $\mathcal{E} \vdash \mathtt{e'} : \tau, \sigma'$. By induction hypothesis with $\mathcal{E}$, $\mathcal{E}' \vdash \mathtt{e} : \tau \xrightarrow{\sigma''} \tau', \sigma$ and $\mathcal{E}' \vdash \mathtt{e'} : \tau, \sigma'$. By definition of the rule (app),

$$\mathcal{E}' \vdash \mathtt{(e\ e')} : \tau', \sigma \cup \sigma' \cup \sigma''$$

Case of (let) By hypothesis, $\mathcal{E}$ and $\mathcal{E}'$ are such that $\tau'' \preceq \mathcal{E}(\mathtt{x})$ implies $\tau'' \preceq \mathcal{E}'(\mathtt{x})$ for every $\mathtt{x} \in Dom(\mathcal{E})$. By hypothesis, $\mathcal{E} \vdash \mathtt{(let\ (x\ e)\ e')} : \tau', \sigma \cup \sigma'$. By definition of the rule (let), this requires that $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ and $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\sigma, \mathcal{E})(\tau)\} \vdash \mathtt{e'} : \tau', \sigma'$. By induction hypothesis on $\mathtt{e}$,

$$\mathcal{E}' \vdash \mathtt{e} : \tau, \sigma$$

Since $\tau'' \preceq \mathcal{E}(\mathtt{x})$ implies $\tau'' \preceq \mathcal{E}'(\mathtt{x})$ for every $\mathtt{x} \in Dom(\mathcal{E})$ and by definition of $\preceq$, $fv(\mathcal{E}') \subseteq fv(\mathcal{E})$. Since $fv(\mathcal{E}') \subseteq fv(\mathcal{E})$ and by definition of Gen, $\tau'' \preceq Gen(\sigma, \mathcal{E})(\tau)$ implies $\tau'' \preceq Gen(\sigma, \mathcal{E}')(\tau)$. By induction hypothesis on $\mathtt{e'}$, we get

$$\mathcal{E}'_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\sigma, \mathcal{E}')(\tau)\} \vdash \mathtt{e'} : \tau', \sigma'$$

By definition of the rule (let), we conclude that

$$\mathcal{E}' \vdash \mathtt{(let\ (x\ e)\ e')} : \tau', \sigma \cup \sigma'$$

Case of (sub) By hypothesis, $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$. By definition of the rule (sub), this requires that there exists $\sigma \supseteq Observe(\mathcal{E}, \tau)(\sigma')$ such that $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'$. Since $\tau'' \preceq \mathcal{E}(\mathtt{x})$ implies $\tau'' \preceq \mathcal{E}'(\mathtt{x})$ for every $\mathtt{x} \in Dom(\mathcal{E})$ and by definition of $\preceq$, $fv(\mathcal{E}') \subseteq fv(\mathcal{E})$. Since $fv(\mathcal{E}') \subseteq fv(\mathcal{E})$ and by definition of $Observe$, $\sigma \supseteq Observe(\mathcal{E}', \tau)(\sigma')$. By definition of the rule (sub),

$$\mathcal{E}' \vdash \mathtt{e} : \tau, \sigma \ \square$$

3.2.1 Conservativity over ML

It is a simple and interesting exercise to show that the type and effect discipline is conservative over Milner's ML typing discipline. This consists of formally proving that any ML expression, that is well-typed in Milner's typing discipline, is also typable in the type and effect discipline.

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{(e\ e')} \mid \mathtt{(lambda\ (x)\ e)} \mid \mathtt{(let\ (x\ e)\ e')} \qquad \text{ML expressions}$$

The syntax of ML expressions can be defined by the above restriction of the syntax of section 2.2.1 which excludes store operators. We also define ML types as a restriction of those presented in section 3.1.

$$\tau ::= int \mid \alpha \mid \tau \xrightarrow{\emptyset} \tau \qquad \text{ML types}$$

Then, the typing discipline of ML can be stated as follows.

$$\frac{\tau \preceq \mathcal{E}(\mathtt{x})}{\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{x} : \tau} \quad (\text{var})_{\text{dm}}$$

$$\frac{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash_{\mathtt{dm}} \mathtt{e} : \tau'}{\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\emptyset} \tau'} \quad (\text{abs})_{\text{dm}}$$

$$\frac{\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{e} : \tau \xrightarrow{\emptyset} \tau' \qquad \mathcal{E} \vdash_{\mathtt{dm}} \mathtt{e'} : \tau}{\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{(e\ e')} : \tau'} \quad (\text{app})_{\text{dm}}$$

$$\frac{\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{e} : \tau \qquad \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\mathcal{E})(\tau)\} \vdash_{\mathtt{dm}} \mathtt{e'} : \tau'}{\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{(let\ (x\ e)\ e')} : \tau'} \quad (\text{let})_{\text{dm}}$$

Static Semantics of ML

Proposition 3.1 (Conservativity over ML) *Let $\mathcal{E}$ a ML type environment and $\mathtt{e}$ an ML expression. If $\mathcal{E} \vdash_{dm} \mathtt{e} : \tau$ then τ is an ML type and $\mathcal{E} \vdash \mathtt{e} : \tau, \emptyset$.*

Proof The proof is by induction on the syntax of expressions.

Case of (var) By hypothesis, $\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{x} : \tau$. By definition of $(\mathrm{var})_{\mathrm{dm}}$, $\tau \preceq \mathcal{E}(\mathtt{x})$. By definition of the rule (var), $\mathcal{E} \vdash \mathtt{x} : \tau, \emptyset$.

Case of (abs) By hypothesis, $\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\emptyset} \tau'$. By definition of $(\mathrm{abs})_{\mathrm{dm}}$, $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash_{\mathtt{dm}} \mathtt{e} : \tau'$. By induction hypothesis on $\mathtt{e}$ with the ML environment $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\}$, we have $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \emptyset$. By definition of the rule (abs),

$$\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\emptyset} \tau', \emptyset$$

Case of (app) By hypothesis, $\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{(e\ e')} : \tau'$. By definition of $(\mathrm{app})_{\mathrm{dm}}$, $\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{e} : \tau \xrightarrow{\emptyset} \tau'$ and $\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{e'} : \tau$. By induction hypothesis on $\mathtt{e}$, $\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\emptyset} \tau', \emptyset$. By induction hypothesis on $\mathtt{e'}$, $\mathcal{E} \vdash \mathtt{e'} : \tau, \emptyset$. By the rule (app),

$$\mathcal{E} \vdash \mathtt{(e\ e')} : \tau', \emptyset$$

Case of (let) By hypothesis, $\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{(let\ (x\ e)\ e')} : \tau'$. By definition of $(\mathrm{let})_{\mathrm{dm}}$, $\mathcal{E} \vdash_{\mathtt{dm}} \mathtt{e} : \tau$ and $\mathcal{E} + \mathit{Gen}(\mathcal{E})(\tau) \vdash_{\mathtt{dm}} \mathtt{e'} : \tau'$. By induction hypothesis on $\mathtt{e}$, we have $\mathcal{E} \vdash \mathtt{e} : \tau, \emptyset$.

We have that $\mathit{Gen}(\emptyset, \mathcal{E})(\tau) = \mathit{Gen}(\mathcal{E})(\tau)$ is an ML type scheme. Thus, by induction hypothesis on $\mathtt{e'}$ with $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\mathcal{E})(\tau)\}$ we get

$$\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\emptyset, \mathcal{E})(\tau)\} \vdash \mathtt{e'} : \tau', \emptyset$$

By definition of the rule (let), we conclude that

$$\mathcal{E} \vdash \mathtt{(let\ (x\ e)\ e')} : \tau', \emptyset \ \square$$

3.2.2 Deterministic Deduction

Finally, and as in the previous chapter 2, a syntax directed inference system can be obtained by the composition of the subsumption rule with the others. Every derivation of $\vdash_d$ can be inductively translated by a derivation in $\vdash$.

$$\frac{\tau \prec \mathcal{E}(\mathtt{x})}{\mathcal{E} \vdash_{\mathtt{d}} \mathtt{x} : \tau, \emptyset} \qquad (\mathrm{var})$$

$$\frac{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash_{\mathtt{d}} \mathtt{e} : \tau', \sigma}{\mathcal{E} \vdash_{\mathtt{d}} \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\sigma \cup \sigma'} \tau', \emptyset} \qquad (\mathrm{abs})+(\mathrm{does})$$

$$\frac{\mathcal{E} \vdash_{\mathtt{d}} \mathtt{e} : \tau \xrightarrow{\sigma} \tau', \sigma' \qquad \mathcal{E} \vdash_{\mathtt{d}} \mathtt{e'} : \tau, \sigma''}{\mathcal{E} \vdash_{\mathtt{d}} \mathtt{(e\ e')} : \tau', \mathit{Observe}(\mathcal{E}, \tau')(\sigma \cup \sigma' \cup \sigma'')} \qquad (\mathrm{app})+(\mathrm{obs})$$

$$\frac{\mathcal{E} \vdash_{\mathtt{d}} \mathtt{e} : \tau, \sigma \qquad \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\mathcal{E})(\tau)\} \vdash_{\mathtt{d}} \mathtt{e'} : \tau', \sigma'}{\mathcal{E} \vdash_{\mathtt{d}} \mathtt{(let\ (x\ e)\ e')} : \tau', \mathit{Observe}(\mathcal{E}, \tau')(\sigma \cup \sigma')} \qquad (\mathrm{let})+(\mathrm{obs})$$

Syntax-Directed Static Semantics

Proposition 3.2 (Syntax-Directed Static Semantics) *If* $\mathcal{E} \vdash_d \mathtt{e} : \tau, \sigma$ *then* $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$. *If* $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma$ *then* $\mathcal{E} \vdash_d \mathtt{e} : \tau, \sigma'$ *for some* $\sigma' \subseteq \sigma$.

Proof Derivations using the rules (var), (app) and (let) are inductively translated by applying the rule (obs) to their conclusion. Derivations using the rule (abs) are translated by using the rules (does) to their premise, as in proposition 2.1 $\square$

3.3 Consistency of Dynamic and Static Semantics

In the static semantics of section 3.1, we introduced type schemes into type environments, types into effects and the inference of observable effects. In order to incorporate these changes, the relation between values and types, that was defined in the chapter 2, must be reformulated.

3.3.1 Consistency Relation

We define a new consistency judgment $s{:}\sigma, \mathcal{S} \models v{:}\tau$ which relates values and types according to a given store model $\mathcal{S}$, a store s and observable effects σ. In the dynamic semantics, when an expression is evaluated, its initial store s possibly mutates to another, s'.

Similarly, in the static semantics, the observable effects σ that correspond to the construction of the initial store s may be augmented with the effect σ', inferred for the evaluated expression. Similarly, the store model $\mathcal{S}$ must be updated to $\mathcal{S}'$. However, $\mathcal{S}'$ must agree with $\mathcal{S}$ on the locations l of its domain which refer to observable regions in σ. This considerations are formalized by the following definition 3.1.

Definition 3.1 (Extension) *$(\sigma', \mathcal{S}')$ extends $(\sigma, \mathcal{S})$, noted $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$ or $(\sigma', \mathcal{S}') \sqsupseteq (\sigma, \mathcal{S})$, iff:*

$$\begin{array}{l} \sigma \subseteq \sigma' \\ Dom(\mathcal{S}) \subseteq Dom(\mathcal{S}') \end{array} \quad \textit{and, for all } l \in Dom(\mathcal{S}), \quad \left\{ \begin{array}{l} \textit{if } \mathcal{S}(l) \in Rng(\sigma) \textit{ then } \mathcal{S}'(l) = \mathcal{S}(l) \\ \textit{if } \mathcal{S}'(l) \in Rng(\sigma) \textit{ then } \mathcal{S}'(l) = \mathcal{S}(l) \end{array} \right.$$

$(\sigma', \mathcal{S}')$ and $(\sigma, \mathcal{S})$ are equivalent, noted $(\sigma, \mathcal{S}) \simeq (\sigma', \mathcal{S}')$, if and only if $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$ and $(\sigma', \mathcal{S}') \sqsubseteq (\sigma, \mathcal{S})$.

The relation, presented in the definition 3.2 below, specifies the consistency between values and types according to observable effects. It refers to an effect which represents the effect of evaluating an expression to a value and the history of the effects that permitted the evaluation of its environment. Unobservable effects may however be needed to show the consistency between unused values, captured within closures, and the types assigned to them.

Example The following example describes this situation. The expression below builds a closure `f` which references a value `y` in the environment it captures. The closure `f` operates upon functions of type $\alpha \xrightarrow{\varsigma} \alpha$ for any α and ς.

```
(let (f (let (y (new (lambda (x) x)))
          (lambda (x)
            (lambda (z) (if true y (new x)) z)
            x)))
  (f (lambda (x) x)))
```

Evaluating the expression `(let (y ...) ...)`, whose value is bound to `f`, does not proceed with observable effects. Thus, the value of `f` and its type are consistent given any store model $\mathcal{S}$ and effect σ. This is not the case, however, with the environment, noted E, captured by the value `f`. It references the value boud to `y`, which is a pointer, noted l, to the identity function. To prove the consistency of the environment E with respect to a type environment, such as $\mathcal{E} = \{\mathtt{y} \mapsto ref_\rho(\alpha \xrightarrow{\varsigma} \alpha)\}$, one must consider some effects, unobservable from outside `f`, related to the pointer l. The appropriate store model $\mathcal{S}'$, equivalent to $\mathcal{S}$ on σ, must relate l to the appropriate region ρ and type $\alpha \xrightarrow{\varsigma} \alpha$ ■

Definition 3.2 (Consistent values and types) *Given the store s, the effects σ and the model $\mathcal{S}$, the consistency relation between a value v and a type τ, written $s{:}\sigma, \mathcal{S} \models v{:}\tau$, satisfies the following property.*

$$\begin{array}{l} s{:}\sigma, \mathcal{S} \models u{:}unit \\ s{:}\sigma, \mathcal{S} \models l{:}ref_\rho(\tau) \Leftrightarrow (\rho, \tau) \in Rng(\sigma),\ \mathcal{S}(l) = (\rho, \tau) \textit{ and } s{:}\sigma, \mathcal{S} \models s(l){:}\tau \\ s{:}\sigma, \mathcal{S} \models (\mathtt{x}, \mathtt{e}, E){:}\tau \Leftrightarrow \textit{there exist } \mathcal{E} \textit{ such that } \mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset, \\ \qquad\qquad\qquad\qquad\qquad\qquad \sigma' \textit{ such that } Observe(\tau)(\sigma') = \emptyset \textit{ and } Observe(\sigma)(\sigma') = \emptyset, \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathcal{S}' \textit{ such that } (\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}') \textit{ and } s{:}\sigma \cup \sigma', \mathcal{S}' \models E{:}\mathcal{E} \end{array}$$

We write $s{:}\sigma, \mathcal{S} \models v{:}\forall\vec{v}.\tau$ *if and only if* $s{:}\sigma, \mathcal{S} \models v{:}\theta\tau$ *for any substitution* θ *defined on* $\vec{v}$ *and* $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$ *if and only if* $Dom(E) = Dom(\mathcal{E})$ *and* $s{:}\sigma, \mathcal{S} \models E(\mathtt{x}){:}\mathcal{E}(\mathtt{x})$ *for any* $\mathtt{x} \in Dom(E)$.

In a similar way as in the chapter 2, we must define the typing consistency relation as the maximal fixed point of the property defined in 3.3. This is done by considering the appropriate function $\mathcal{F}$ below.

Definition 3.3 (F) *The function* $\mathcal{F}$ *is defined over the elements* $\mathcal{Q}$ *of* $\mathcal{P}(\mathcal{R})$. $\mathcal{R}$ *is the set of all* $(s, \sigma, \mathcal{S}, v, \tau)$. *The greatest fixed point of* $\mathcal{F}$, $gfp(\mathcal{F}) = \cup\{\mathcal{Q} \subseteq \mathcal{S} \mid \mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})\}$, *defines our relation: we write* $s, \sigma, \mathcal{S} \models v{:}\tau$ *if and only if* $(s, \sigma, \mathcal{S}, v, \tau) \in gfp(\mathcal{F})$.

$$
\begin{array}{l}
\mathcal{F}(\mathcal{Q})=\{(s, \sigma, \mathcal{S}, v, \tau) \mid \\
\quad \textit{if } v = u \textit{ then } \tau = \textit{unit} \\
\quad \textit{if } v = l \textit{ then } \tau = ref_\rho(\tau'),\ (\rho, \tau') \in Rng(\sigma),\ \mathcal{S}(l) = (\rho, \tau') \textit{ and } (s, \sigma, \mathcal{S}, s(l), \tau') \in \mathcal{Q} \\
\quad \textit{if } v = (\mathtt{x}, \mathtt{e}, E) \textit{ then there exist } \mathcal{E} \textit{ such that } \mathcal{E} \vdash \texttt{(lambda (x) e)} : \tau, \emptyset, \\
\qquad\qquad\qquad\qquad \sigma' \textit{ such that } Observe(\tau)(\sigma') = \emptyset \textit{ and } Observe(\sigma)(\sigma') = \emptyset, \\
\qquad\qquad\qquad\qquad \mathcal{S}' \textit{ such that } (\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}') \\
\qquad\qquad \textit{and that } (s, \sigma \cup \sigma', \mathcal{S}', E(\mathtt{x}), \tau') \in \mathcal{Q} \textit{ for any } \mathtt{x} \in Dom(E) \textit{ and } \tau' \preceq \mathcal{E}(\mathtt{x})\}
\end{array}
$$

To admit a maximal fixed point $gfp(\mathcal{F})$, the function $\mathcal{F}$ must be monotonic. This is the first property that we thus have to verify.

Lemma 3.6 (Monotony of $\mathcal{F}$) *If* $\mathcal{Q} \subseteq \mathcal{Q}'$ *then* $\mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$.

Proof Let $\mathcal{Q}$ and $\mathcal{Q}'$ be two subsets of $\mathcal{S}$ such that $\mathcal{Q} \subseteq \mathcal{Q}'$. Let q be $(s, \sigma, \mathcal{S}, v, \tau)$ in $\mathcal{F}(Q)$. We prove that $q \in \mathcal{F}(\mathcal{Q}')$.

- If $v = u$ then, by the definition 3.3, $\tau = \textit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q}')$.
- If $v = l$ then, by the definition 3.3, $\tau = ref_\rho(\tau')$, $(\rho, \tau') \in Rng(\sigma)$, $\mathcal{S}(l) = (\rho, \tau')$ and $(s, \sigma, \mathcal{S}, s(l), \tau') \in \mathcal{Q}$. Since $\mathcal{Q} \subseteq \mathcal{Q}'$, $(s, \sigma, \mathcal{S}, s(l), \tau') \in \mathcal{Q}'$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q}')$.
- If $v = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 3.3, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset$, σ' such that $Observe(\tau)(\sigma') = \emptyset$ and $Observe(\sigma)(\sigma') = \emptyset$, $\mathcal{S}'$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}')$ and $(s, \sigma \cup \sigma', \mathcal{S}', E(\mathtt{x}), \tau') \in \mathcal{Q}$ for any $\mathtt{x} \in Dom(E)$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$. Since $\mathcal{Q} \subseteq \mathcal{Q}'$, $(s, \sigma \cup \sigma', \mathcal{S}', E(\mathtt{x}), \tau') \in \mathcal{Q}'$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q}')$ □

The lemma 3.7 connects the definition of the relation $\sqsubseteq$ in 3.1 with the consistency relation, defined in 3.2, according to the following respects.

Lemma 3.7 (Extension) *If* $s{:}\sigma, \mathcal{S} \models v{:}\tau$ *and* $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$ *then* $s{:}\sigma', \mathcal{S}' \models v{:}\tau$.

Proof We consider the set $\mathcal{Q}=\{(s, \sigma', \mathcal{S}', v, \tau) \mid s{:}\sigma, \mathcal{S} \models v{:}\tau$ and $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')\}$. We show, by case analysis on the structure of v, that $q = (s, \sigma', \mathcal{S}', v, \tau)$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v = u$ then, by the definition 3.2, $\tau = \textit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.
- If $v = l$ then, by the definition 3.2, $\tau = ref_\rho(\tau')$, $(\rho, \tau') \in Rng(\sigma)$, $\mathcal{S}(l) = (\rho, \tau')$ and $s{:}\sigma, \mathcal{S} \models s(l){:}\tau'$. Since $s{:}\sigma, \mathcal{S} \models s(l){:}\tau'$ and by definition of $\mathcal{Q}$, $(s, \sigma', \mathcal{S}', s(l), \tau') \in \mathcal{Q}$. Since $(\rho, \tau') \in Rng(\sigma)$, $\mathcal{S}(l) = (\rho, \tau')$ and by definition 3.1, $(\rho, \tau') \in Rng(\sigma')$ and $\mathcal{S}'(l) = (\rho, \tau')$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$.
- If $v = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 3.2, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset$, σ_1 such that $Observe(\tau)(\sigma_1) = \emptyset$ and $Observe(\sigma)(\sigma_1) = \emptyset$, $\mathcal{S}_1$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $s{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E(\mathtt{x}){:}\tau'$ for any $\mathtt{x} \in Dom(E)$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$.

  Let us define θ on $Regs(\sigma_1)$ in such a way that the regions $Regs(\theta\sigma_1)$ are not free in τ, σ' and $\mathcal{S}'$. Let us write $\mathcal{E}' = \theta\mathcal{E}$, $\sigma'_1 = \theta\sigma_1$ and $\mathcal{S}'_1 = \theta\mathcal{S}_1$. By the lemma 3.4, $\mathcal{E}' \vdash$ `(lambda (x) e)` $: \tau, \emptyset$. By

the lemma 3.1, for all $\tau'' \preceq \mathcal{E}'(\mathtt{x})$, there exists $\tau' \preceq \mathcal{E}(\mathtt{x})$ such that the restriction θ' of θ on the free variables of $\mathcal{E}$ verifies $\tau'' = \theta'\tau'$. By the lemma 3.9, $s{:}\sigma \cup \sigma'_1, \mathcal{S}'_1 \models E(\mathtt{x}){:}\tau''$ for any $\mathtt{x} \in Dom(E)$ and $\tau'' \preceq \mathcal{E}'(\mathtt{x})$. Let us define $\mathcal{S}''_1$ as follows,

$$\forall l \in Dom(\mathcal{S}'),\ \mathcal{S}''_1(l) = \begin{cases} \mathcal{S}'_1(l), & \text{if } l \in Dom(\mathcal{S}) \text{ and } \mathcal{S}'_1(l) \in Rng(\sigma \cup \sigma'_1) \\ \mathcal{S}'(l) & \text{otherwise} \end{cases}$$

To prove that $(s, \sigma' \cup \sigma'_1, \mathcal{S}''_1, E(\mathtt{x}), \tau'') \in \mathcal{Q}$, it remains to show that $(\sigma \cup \sigma'_1, \mathcal{S}'_1) \sqsubseteq (\sigma \cup \sigma'_1, \mathcal{S}''_1)$. This requires that, for any $l \in Dom(\mathcal{S}'_1)$, if $\mathcal{S}'_1(l) \in Rng(\sigma \cup \sigma'_1)$ or $\mathcal{S}''_1(l) \in Rng(\sigma \cup \sigma'_1)$ then $\mathcal{S}''_1(l) = \mathcal{S}'_1(l)$.

- If $\mathcal{S}'_1(l) \in Rng(\sigma \cup \sigma'_1)$ then, by definition of $\mathcal{S}''_1$, $\mathcal{S}''_1(l) = \mathcal{S}'_1(l)$.
- If $\mathcal{S}''_1(l) \in Rng(\sigma \cup \sigma'_1)$, we proceed by case analysis on the definition of $\mathcal{S}''_1$. First, if $l \in Dom(\mathcal{S})$ and $\mathcal{S}'_1(l) \in Rng(\sigma \cup \sigma'_1)$ then $\mathcal{S}''_1(l) = \mathcal{S}'_1(l)$. Otherwise, either $l \notin Dom(\mathcal{S})$ [This is impossible, since we suppose that $l \in Dom(\mathcal{S}'_1)$] or $\mathcal{S}'_1(l) \notin Rng(\sigma \cup \sigma'_1)$. We show that this is impossible as well.

  By hypothesis, we have that $\mathcal{S}''_1(l) = \mathcal{S}'(l) \in Rng(\sigma \cup \sigma'_1)$. Since, by definition of σ'_1, $fr(\Im(\mathcal{S}')) \cap Regs(\sigma'_1) = \emptyset$, we must have $\mathcal{S}''_1(l) = \mathcal{S}'(l) \in Rng(\sigma)$. However, by hypothesis, we have that $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$. This imposes that, if $\mathcal{S}(l)$ or $\mathcal{S}'(l)$ is in $Rng(\sigma)$, then $\mathcal{S}'(l) = \mathcal{S}(l)$. Since $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and θ does not affect σ, we must have $\mathcal{S}'(l) = \mathcal{S}(l) = \mathcal{S}_1(l) = \mathcal{S}'_1(l)$. Thus, we cannot have both $\mathcal{S}'(l) \in Rng(\sigma)$ and $\mathcal{S}'_1(l) \notin Rng(\sigma \cup \sigma'_1)$.

We have shown, by the definition 3.1, that $(\sigma \cup \sigma'_1, \mathcal{S}'_1) \sqsubseteq (\sigma' \cup \sigma'_1, \mathcal{S}''_1)$. Since, for any $\mathtt{x} \in Dom(E)$ and $\tau'' \preceq \mathcal{E}'(\mathtt{x})$, $s{:}\sigma \cup \sigma'_1, \mathcal{S}'_1 \models E(\mathtt{x}){:}\tau''$, by definition of $\mathcal{Q}$, $(s, \sigma' \cup \sigma'_1, \mathcal{S}''_1, E(\mathtt{x}), \tau'') \in \mathcal{Q}$.

To show that $q \in \mathcal{F}(\mathcal{Q})$, we need to prove that $(\sigma', \mathcal{S}') \simeq (\sigma', \mathcal{S}''_1)$. Since $Dom(\mathcal{S}') = Dom(\mathcal{S}''_1)$, it remains to show that, if $\mathcal{S}'(l) \in Rng(\sigma')$ or $\mathcal{S}''_1(l) \in Rng(\sigma')$, then $\mathcal{S}'(l) = \mathcal{S}''_1(l)$. We proceed by case analysis on σ and $\sigma' \setminus \sigma$.

- If $\mathcal{S}''_1(l) \in Rng(\sigma)$ or $\mathcal{S}'(l) \in Rng(\sigma)$ then either $l \in Dom(\mathcal{S})$ or not. If $l \in Dom(\mathcal{S})$ then, since $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$, $\mathcal{S}(l) = \mathcal{S}_1(l)$. Since θ is not defined on σ, $\mathcal{S}_1(l) = \mathcal{S}'_1(l)$. By definition of $\mathcal{S}''_1$, $\mathcal{S}''_1(l) = \mathcal{S}'_1(l)$. Since, by hypothesis, $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$, $\mathcal{S}(l) = \mathcal{S}'(l)$. Thus, $\mathcal{S}''_1(l) = \mathcal{S}'(l)$. If $l \notin Dom(\mathcal{S})$ then, by definition of $\mathcal{S}''_1$, $\mathcal{S}''_1(l) = \mathcal{S}'(l)$. Thus, if $\mathcal{S}'(l) \in Rng(\sigma)$ then $\mathcal{S}'(l) = \mathcal{S}''_1(l)$.
- If $\mathcal{S}''_1(l) \in Rng(\sigma' \setminus \sigma)$ then, since $Regs(\sigma'_1) \cap fr(\sigma') = \emptyset$, $\mathcal{S}''_1(l) \notin Rng(\sigma \cup \sigma'_1)$. Thus, by definition of $\mathcal{S}''_1$, $\mathcal{S}''_1(l) = \mathcal{S}'(l)$. Similarly, if $\mathcal{S}'(l) \in Rng(\sigma' \setminus \sigma)$ then, since $Regs(\sigma'_1) \cap fr(\sigma') = \emptyset$, $\mathcal{S}'(l) \notin Rng(\sigma \cup \sigma'_1)$. Thus, either $l \notin Dom(\mathcal{S})$ and then $\mathcal{S}'(l) = \mathcal{S}''_1(l)$, or $l \in Dom(\mathcal{S})$ and then, since $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$, $\mathcal{S}(l) \notin Rng(\sigma)$. Thus, $\mathcal{S}'_1(l) \notin Rng(\sigma)$ and, by definition of $\mathcal{S}''_1$, $\mathcal{S}''_1(l) = \mathcal{S}'(l)$.

We have proved, by the definition 3.1, that $(\sigma', \mathcal{S}') \simeq (\sigma', \mathcal{S}''_1)$. By the definition of σ'_1, $Observe(\tau)(\sigma'_1) = \emptyset$ and $Observe(\sigma')(\sigma'_1) = \emptyset$. Since $\mathcal{E}' \vdash$ `(lambda (x) e)` $: \tau, \emptyset$ and $(s, \sigma' \cup \sigma'_1, \mathcal{S}''_1, E(\mathtt{x}), \tau'') \in \mathcal{Q}$ for every $\mathtt{x} \in Dom(E)$ and every $\tau'' \preceq \mathcal{E}'(\mathtt{x})$, by the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$ □

The lemma 3.8 connects the definition 3.1 of the relation $\simeq$ with the consistency relation, defined in 3.2, according to the following respects.

Lemma 3.8 (Equivalence) *If $(\sigma, \mathcal{S}) \simeq (\sigma', \mathcal{S}')$ then $s{:}\sigma, \mathcal{S} \models v{:}\tau$ if and only if $s{:}\sigma', \mathcal{S}' \models v{:}\tau$.*

Proof By hypothesis, $(\sigma, \mathcal{S}) \simeq (\sigma', \mathcal{S}')$. By definition 3.1, $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$ and $(\sigma, \mathcal{S}) \sqsupseteq (\sigma', \mathcal{S}')$. If $s{:}\sigma, \mathcal{S} \models v{:}\tau$ then, since $(\sigma, \mathcal{S}) \sqsubseteq (\sigma', \mathcal{S}')$ and by the lemma 3.7, $s{:}\sigma', \mathcal{S}' \models v{:}\tau$. If $s{:}\sigma', \mathcal{S}' \models v{:}\tau$ then, since $(\sigma, \mathcal{S}) \sqsupseteq (\sigma', \mathcal{S}')$ and by the lemma 3.7, $s{:}\sigma, \mathcal{S} \models v{:}\tau$ □

The lemma 3.9 states that the typing judgment $s{:}\sigma, \mathcal{S} \models v{:}\tau$ is stable under substitution.

Lemma 3.9 (Substitution) *If $s{:}\sigma, \mathcal{S} \models v{:}\tau$ then $s{:}\theta\sigma, \theta\mathcal{S} \models v{:}\theta\tau$ for any substitution θ.*

Proof We consider the set $\mathcal{Q}$=$\{(s,\theta\sigma,\theta\mathcal{S},v,\theta\tau) \mid s{:}\sigma,\mathcal{S} \models v{:}\tau\}$. We show, by case analysis on the structure of v, that $q=(s,\theta\sigma,\theta\mathcal{S},v,\theta\tau)$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v=u$ then, by the definition 3.2, $\tau = \mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.
- If $v=l$ then, by the definition 3.2, $\tau = \mathit{ref}_\rho(\tau')$, $(\rho,\tau') \in \mathit{Rng}(\sigma)$, $\mathcal{S}(l)=(\rho,\tau')$ and $s{:}\sigma,\mathcal{S} \models s(l),\tau'$. Since $s{:}\sigma,\mathcal{S} \models s(l){:}\tau'$ and by definition of $\mathcal{Q}$, $(s,\theta\sigma,\theta\mathcal{S},s(l),\theta\tau') \in \mathcal{Q}$, $\theta(\rho,\tau') = \theta\mathcal{S}(l) \in \mathit{Rng}(\theta\sigma)$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$.
- If $v=(\mathtt{x},\mathtt{e},E)$ then, by the definition 3.2, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau,\emptyset,\sigma'$ such that $\mathit{Observe}(\tau)(\sigma')=\emptyset$ and $\mathit{Observe}(\sigma)(\sigma')=\emptyset$, $\mathcal{S}'$ such that $(\sigma,\mathcal{S}) \simeq (\sigma,\mathcal{S}')$ and $s{:}\sigma\cup\sigma',\mathcal{S}' \models E(\mathtt{x}){:}\tau'$ for any $\mathtt{x} \in \mathit{Dom}(E)$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$. Let us define the substitution θ' of $\mathit{Regs}(\sigma')$ by regions $\mathit{Regs}(\theta'\sigma')$ not free in $\theta\tau$ or $\theta\sigma$ and let θ' be same as θ elsewhere.

  Let us write $\mathcal{E}'=\theta'\mathcal{E}$, $\mathcal{S}''=\theta'\mathcal{S}'$ and $\sigma''=\theta'\sigma'$. By the lemma 3.1, for all $\tau'' \preceq \mathcal{E}'(\mathtt{x})$, there exists $\tau' \preceq \mathcal{E}(\mathtt{x})$ such that the restriction θ'' of θ' on the free variables of $\mathcal{E}$ verifies $\tau''=\theta''\tau'$. By definition of $\mathcal{Q}$, $(s,\theta\sigma\cup\sigma'',\mathcal{S}'',E(\mathtt{x}),\tau'') \in \mathcal{Q}$ for every $\mathtt{x} \in \mathit{Dom}(E)$ and every $\tau'' \preceq \mathcal{E}'(\mathtt{x})$. By definition of *Observe* and θ', $\mathit{Observe}(\theta\sigma)(\sigma'')=\emptyset$ and $\mathit{Observe}(\theta\tau)(\sigma'')=\emptyset$. By the definition 3.1, $(\theta\sigma,\theta\mathcal{S}) \simeq (\theta\sigma,\mathcal{S}'')$. By the lemma 3.4 with θ', $\mathcal{E}' \vdash$ `(lambda (x) e)` $: \theta\tau,\emptyset$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$ □

The lemma 3.10 is a refinement of lemma 3.9 and states that the judgment $\mathcal{S} \models v{:}\tau$ reduces to $\mathcal{S} \models v{:}\theta\tau$ for any substitution θ that affects τ but not σ. It is used in the proof of consistency, theorem 3.3, to show that our type generalization criterion is correct.

Lemma 3.10 (Instantiation) *If $s{:}\sigma,\mathcal{S} \models v{:}\tau$ and θ is defined on $fv(\tau) \setminus fv(\sigma)$ then $s{:}\sigma,\mathcal{S} \models v{:}\theta\tau$.*

Proof By hypothesis, $s{:}\sigma,\mathcal{S} \models v{:}\tau$ and θ is defined on $fv(\tau) \setminus fv(\sigma)$. By the lemma 3.9, $s{:}\sigma,\theta\mathcal{S} \models v{:}\theta\tau$. By the definition 3.1, $(\sigma,\mathcal{S}) \simeq (\sigma,\theta\mathcal{S})$. By the lemma 3.8, $s{:}\sigma,\mathcal{S} \models v{:}\theta\tau$□

3.3.2 Notion of Succession

During the evaluation of an expression, the store is extended and updated in an organized way. The definition 3.4 specifies the requirements for preserving consistency between types and values in the presence of side-effects.

Definition 3.4 (Succession) *$(s,\sigma,\mathcal{S})$ becomes $(s',\sigma',\mathcal{S}')$, noted $(s,\sigma,\mathcal{S}) \sqsubseteq (s',\sigma',\mathcal{S}')$, if and only if $\mathit{Dom}(s) \subseteq \mathit{Dom}(s')$, $(\sigma,\mathcal{S}) \sqsubseteq (\sigma',\mathcal{S}')$ and $s{:}\sigma,\mathcal{S} \models v{:}\tau$ implies $s'{:}\sigma',\mathcal{S}' \models v{:}\tau$ for any v and τ.*

The lemma 3.11 represents the situation that arises when a reference is initialized. In the lemma 3.12, we address the situation arising when a value is assigned to a reference.

Lemma 3.11 (Initialization) *Let $\sigma'=\mathit{init}(\rho,\tau)$, $s'=s+\{l \mapsto v\}$ ($l \notin \mathit{Dom}(s)$), $\mathcal{S}'=\mathcal{S}+\{l \mapsto (\rho,\tau)\}$ ($l \notin \mathit{Dom}(\mathcal{S})$) and $s{:}\sigma,\mathcal{S} \models v{:}\tau$. If $s{:}\sigma,\mathcal{S} \models v'{:}\tau'$ then $s'{:}\sigma\cup\sigma',\mathcal{S}' \models v'{:}\tau'$.*

Proof We consider the set $\mathcal{Q}$=$\{(s',\sigma\cup\sigma',\mathcal{S}',v',\tau') \mid s{:}\sigma,\mathcal{S} \models v'{:}\tau'\}$. We show, by case analysis on the structure of v, that $q=(s',\sigma\cup\sigma',\mathcal{S}',v',\tau')$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v'=u$ then, by the definition 3.2, $\tau'=\mathit{unit}$ so that $q \in \mathcal{F}(\mathcal{Q})$.
- If $v'=l'$ then, by the definition 3.2, $\tau'=\mathit{ref}_{\rho'}(\tau'')$, $(\rho',\tau'') \in \mathit{Rng}(\sigma)$, $\mathcal{S}(l')=(\rho',\tau'')$ and $s{:}\sigma,\mathcal{S} \models s(l'),\tau''$. Since $s{:}\sigma,\mathcal{S} \models s(l'){:}\tau''$, by definition of $\mathcal{Q}$, $(s',\sigma\cup\sigma',\mathcal{S}',s(l'),\tau'') \in \mathcal{Q}$.

  Since $(\rho',\tau'') \in \mathit{Rng}(\sigma)$ and $\mathcal{S}(l')=(\rho',\tau'')$ then, by the definition 3.1, $(\rho',\tau'') \in \mathit{Rng}(\sigma\cup\sigma')$ and $\mathcal{S}'(l')=(\rho',\tau'')$. If $l'=l$ then, by hypothesis, $v=s'(l')$, $\tau''=\tau$ and $s{:}\sigma,\mathcal{S} \models v{:}\tau$. Thus, by definition of $\mathcal{Q}$, $(s',\sigma\cup\sigma',\mathcal{S}',s'(l'),\tau) \in \mathcal{Q}$. Otherwise, $s'(l')=s(l')$ and $(s',\sigma\cup\sigma',\mathcal{S}',s(l'),\tau'') \in \mathcal{Q}$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$.

- If $v' = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 3.2, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau', \emptyset, \sigma_1$ such that $Observe(\tau')(\sigma_1) = \emptyset$ and $Observe(\sigma)(\sigma_1) = \emptyset$, $\mathcal{S}_1$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $s{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E(\mathtt{x}){:}\tau_1$ for any $\mathtt{x} \in Dom(E)$ and $\tau_1 \preceq \mathcal{E}(\mathtt{x})$. Let θ be defined on $Regs(\sigma_1)$ and such that the regions $Regs(\theta\sigma_1)$ are not free in τ', σ and σ'. Let $\sigma'_1 = \theta\sigma_1$, $\mathcal{S}'_1 = \theta\mathcal{S}_1$ and $\mathcal{E}' = \theta\mathcal{E}$. Since, by the lemma 3.2, $Regs(\sigma_1) \cap fv(\tau') = \emptyset$, by the lemma 3.4 with θ, $\mathcal{E}' \vdash$ `(lambda (x) e)` $: \tau', \emptyset$.

  By the lemma 3.1, for all $\tau'_1 \preceq \mathcal{E}'(\mathtt{x})$, there exists $\tau_1 \preceq \mathcal{E}(\mathtt{x})$ such that the restriction θ_1 of θ on $\mathcal{E}$ verifies $\tau'_1 = \theta_1\tau_1$. Thus, by the lemma 3.9, $s{:}\sigma \cup \sigma'_1, \mathcal{S}'_1 \models E(\mathtt{x}){:}\tau'_1$ for any $\mathtt{x} \in Dom(E)$ and $\tau'_1 \preceq \mathcal{E}'(\mathtt{x})$. By definition of $\mathcal{Q}$, $(s, \sigma \cup \sigma' \cup \sigma'_1, \mathcal{S}'_1, E(\mathtt{x}), \tau'_1) \in \mathcal{Q}$ for every $\mathtt{x} \in Dom(E)$ and every $\tau'_1 \preceq \mathcal{E}'(\mathtt{x})$. We have defined σ'_1 such that $Observe(\sigma \cup \sigma')(\sigma'_1) = \emptyset$ and $Observe(\tau')(\sigma'_1) = \emptyset$ and $\mathcal{S}'_1$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}'_1)$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$ □

Lemma 3.12 (Assignment) *Let $\sigma' = write(\rho, \tau)$, $s' = s_l + \{l \mapsto v\}$, $s{:}\sigma, \mathcal{S} \models l{:}ref_\rho(\tau)$ and $s{:}\sigma, \mathcal{S} \models v{:}\tau$. If $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$ then $s'{:}\sigma \cup \sigma', \mathcal{S} \models v'{:}\tau'$.*

Proof We consider the set $\mathcal{Q}$=$\{(s', \sigma \cup \sigma', \mathcal{S}, v', \tau') \mid s{:}\sigma, \mathcal{S} \models v'{:}\tau'\}$. We show, by case analysis on the structure of v, that $q = (s', \sigma \cup \sigma', \mathcal{S}, v', \tau')$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v' = u$ then, by the definition 3.2, $\tau' = unit$ so that $q \in \mathcal{F}(\mathcal{Q})$.
- If $v' = l'$ then, by the definition 3.2, $\tau' = ref_{\rho'}(\tau'')$, $(\rho', \tau'') \in Rng(\sigma)$, $\mathcal{S}(l') = (\rho', \tau'')$ and $s{:}\sigma, \mathcal{S} \models s(l'), \tau''$. By definition of $\mathcal{Q}$, $(s, \sigma \cup \sigma', \mathcal{S}, s(l'), \tau'') \in \mathcal{Q}$. Since $\mathcal{S}(l') = (\rho', \tau'')$, by definition of $\mathcal{S}'$, $\mathcal{S}'(l') = (\rho', \tau'')$. Since $(\rho', \tau'') \in Rng(\sigma)$, by definition of σ', $(\rho', \tau'') \in Rng(\sigma \cup \sigma')$. If $l' = l$ then $v = s'(l')$, $s{:}\sigma, \mathcal{S} \models l{:}ref_\rho(\tau)$ and $s{:}\sigma, \mathcal{S} \models v{:}\tau$. Thus, by definition of $\mathcal{Q}$, $(s', \sigma \cup \sigma', \mathcal{S}', v, \tau) \in \mathcal{Q}$. Otherwise, $s'(l') = s(l')$ and $(s', \sigma \cup \sigma', \mathcal{S}', s(l'), \tau'') \in \mathcal{Q}$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$.
- If $v' = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 3.2, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset, \sigma_1$ such that $Observe(\sigma)(\sigma_1) = \emptyset$ and $Observe(\tau)(\sigma_1) = \emptyset$, $\mathcal{S}_1$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $s{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E(\mathtt{x}){:}\tau'$ for any $\mathtt{x} \in Dom(E)$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$. Since $s{:}\sigma, \mathcal{S} \models l{:}ref_\rho(\tau)$, by definition 3.2, $(\rho, \tau) \in Rng(\sigma)$ and then $Rng(\sigma') \subseteq Rng(\sigma)$. Thus, $Observe(\sigma \cup \sigma')(\sigma_1) = \emptyset$. By definition of $\mathcal{Q}$, $(s, \sigma \cup \sigma' \cup \sigma_1, \mathcal{S}_1, E(\mathtt{x}), \tau') \in \mathcal{Q}$ for every $\mathtt{x} \in Dom(E)$ and every $\tau' \preceq \mathcal{E}(\mathtt{x})$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$ □

The lemma 3.13 is used in the inner proof case of the theorem 3.3 in the case the rule (sub) is used. It tells that the consistency of the rest of the computation is not affected by unobservable effects.

Lemma 3.13 (Observability) *If $s{:}\sigma, \mathcal{S} \models v{:}\tau$ and $Observe(\tau)(\sigma) \subseteq \sigma' \subseteq \sigma$ then $s{:}\sigma', \mathcal{S} \models v{:}\tau$.*

Proof We consider the set $\mathcal{Q}$=$\{(s, \sigma', \mathcal{S}, v, \tau) \mid s{:}\sigma, \mathcal{S} \models v{:}\tau,\ \sigma' = Observe(\tau)(\sigma)\}$. For any σ' such that $Observe(\tau)(\sigma) \subseteq \sigma' \subseteq \sigma$ we show, by case analysis on the structure of v, that $q = (s, \sigma', \mathcal{S}, v, \tau)$ is in $\mathcal{F}(\mathcal{Q})$. Thus, $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$, showing that $\mathcal{Q}$ is $\mathcal{F}$-consistent.

- If $v = u$ then, by the definition 3.2, $\tau = unit$ so that $q \in \mathcal{F}(\mathcal{Q})$.
- If $v = l$ then, by the definition 3.2, $\tau = ref_\rho(\tau')$, $(\rho, \tau') \in Rng(\sigma)$, $\mathcal{S}(l) = (\rho, \tau')$ and $s{:}\sigma, \mathcal{S} \models s(l), \tau'$. Since $s{:}\sigma, \mathcal{S} \models s(l){:}\tau'$, by definition of $\mathcal{Q}$, $(s, \sigma', \mathcal{S}, s(l), \tau') \in \mathcal{Q}$. Since $(\rho, \tau') \in Rng(\sigma)$ and $\tau = ref_\rho(\tau')$, by definition of $Observe$, $(\rho, \tau') \in Rng(Observe(\tau)(\sigma))$. Since $Observe(\tau)(\sigma) \subseteq \sigma'$, $(\rho, \tau') \in Rng(\sigma')$. Thus, by the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$.
- If $v = (\mathtt{x}, \mathtt{e}, E)$ then, by the definition 3.2, there exist $\mathcal{E}$ such that $\mathcal{E} \vdash$ `(lambda (x) e)` $: \tau, \emptyset, \sigma_1$ such that $Observe(\tau)(\sigma_1) = \emptyset$ and $Observe(\sigma)(\sigma_1) = \emptyset$, $\mathcal{S}_1$ such that $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $s{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E{:}\mathcal{E}$. By the definition 3.2, for any $\mathtt{x} \in Dom(\mathcal{E})$ and $\tau' \preceq \mathcal{E}(\mathtt{x})$, $s{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E(\mathtt{x}){:}\tau'$. By definition of $\mathcal{Q}$, $(s, \sigma \cup \sigma_1, \mathcal{S}_1, E(\mathtt{x}), \tau') \in \mathcal{Q}$

  Let $\sigma'_1 = \sigma_1 \cup (\sigma \backslash \sigma')$. Since $Observe(\tau)(\sigma_1) = \emptyset$ and $fr(\tau) \cap Regs(\sigma \backslash \sigma') = \emptyset$, we have $Observe(\tau)(\sigma'_1) = \emptyset$. Similarly, since $Observe(\sigma)(\sigma_1) = \emptyset$ and $fr(\sigma') \cap Regs(\sigma \setminus \sigma') = \emptyset$, we have $Observe(\sigma')(\sigma'_1) = \emptyset$. We have that $\sigma \cup \sigma_1 = \sigma' \cup \sigma'_1$ and finally, since $(\sigma, \mathcal{S}) \simeq (\sigma, \mathcal{S}_1)$ and $\sigma' \subseteq \sigma$, $(\sigma', \mathcal{S}) \simeq (\sigma', \mathcal{S}_1)$. By the definition 3.3, $q \in \mathcal{F}(\mathcal{Q})$ □

3.3.3 Consistency Theorem

The consistency theorem appears below. The effect σ corresponds to the effect of evaluating the environment of the expression $\mathtt{e}$. Let E and $\mathcal{E}$ be consistent with respect to this initial effect σ, the initial store s and a store model $\mathcal{S}$ e.g. $s{:}\sigma \cup \sigma', \mathcal{S} \models E{:}\mathcal{E}$. If $\mathtt{e}$ evaluates to $s, E \vdash \mathtt{e} \rightarrow v, f, s'$ and has type and effect $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'$, then there exists a store model $\mathcal{S}'$ such that $(s, \sigma, \mathcal{S})$ becomes $(s', \sigma', \mathcal{S}')$ and that the value v is consistent with its type, according to the model $\mathcal{S}'$ e.g. $s'{:}\sigma', \mathcal{S}' \models v{:}\tau$.

Theorem 3.1 (Consistency of dynamic and static semantics) *If* $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$, $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'$ *and* $s, E \vdash \mathtt{e} \rightarrow v, f, s'$ *then there exists* $\mathcal{S}'$ *such that* $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$ *and* $s'{:}\sigma \cup \sigma', \mathcal{S}' \models v{:}\tau$.

Proof The proof is by induction on the length of the dynamic evaluation. The dynamic trace f is not taken into consideration in this theorem. However, it can be related to the effect σ' modulo the store model $\mathcal{S}'$ by observing that $\mathit{Observe}(\mathcal{E}, \tau)(\mathcal{S}'(f)) \subseteq \sigma'$.

Before detailling the case analysis that corresponds to each syntactic form, we detail the inner proof case that corresponds to the application of the rule (sub) in the static semantics.

The situation is that $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'$, $s, E \vdash \mathtt{e} \rightarrow v, f, s'$ and $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The judgment $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'$ was inferred from the rule of observation.

$$\frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma_1 \qquad \mathit{Observe}(\mathcal{E}, \tau)(\sigma_1) \subseteq \sigma'}{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'}$$

Let us write $\sigma'_1 = \mathit{Observe}(\mathcal{E}, \tau)(\sigma_1)$ and $\sigma'_2 = \sigma_1 \setminus \sigma'_1$. Let us define the substitution θ on $\mathit{Regs}(\sigma'_2)$ such that the regions $\mathit{Regs}(\theta\sigma'_2)$ are not free in $\mathcal{E}$, $\sigma \cup \sigma'_1$ and τ. Let us write $\sigma''_2 = \theta\sigma'_2$. Since $\mathit{Observe}(\mathcal{E}, \tau)(\sigma'_2) = \emptyset$, by the lemma 3.2, $\mathit{Regs}(\sigma'_2) \cap (\mathit{fr}(\mathcal{E}) \cup \mathit{fr}(\tau)) = \emptyset$. By the lemma 3.4,

$$\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'_1 \cup \sigma''_2$$

By hypothesis on the inner proof, there exists $\mathcal{S}'$ such that $s'{:}\sigma \cup \sigma'_1 \cup \sigma''_2, \mathcal{S}' \models v{:}\tau$ and that $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma'_1 \cup \sigma''_2, \mathcal{S}')$. Since $\mathit{Observe}(\mathcal{E}, \tau)(\sigma''_2) = \emptyset$, by the lemma 3.13,

$$s'{:}\sigma \cup \sigma'_1, \mathcal{S}' \models v{:}\tau$$

In the same manner, let us consider any v' and τ' such that $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$. Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma'_1 \cup \sigma''_2, \mathcal{S}')$, by the definition 3.4, $s'{:}\sigma \cup \sigma'_1 \cup \sigma''_2, \mathcal{S}' \models v'{:}\tau'$. We can freely choose $\sigma''_2 = \theta\sigma'_2$, in the judgment $\mathcal{E} \vdash \mathtt{e} : \tau, \sigma'_1 \cup \sigma''_2$, so that $\mathit{Regs}(\sigma''_2) \cap \mathit{fr}(\tau') = \emptyset$. By the definition of *Observe*, $\mathit{Observe}(\tau')(\sigma''_2) = \emptyset$ and thus, by the lemma 3.13, $s'{:}\sigma \cup \sigma'_1, \mathcal{S}' \models v'{:}\tau'$. This holds for any judgment $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$. Thus, by the definition 3.1,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma'_1, \mathcal{S}')$$

Since $\sigma'_1 \subseteq \sigma'$, by the lemma 3.7,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models v{:}\tau$$

Case of (var) By hypothesis $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$, $s, E \vdash \mathtt{x} \rightarrow v, \emptyset, s$ and $\sigma, \mathcal{E} \vdash \mathtt{x} : \tau, \emptyset$. By definition of the rule (var) this requires that $E(\mathtt{x}) = v$ and that $\tau \preceq \mathcal{E}(\mathtt{x})$. By the definition 3.2, $s{:}\sigma, \mathcal{S} \models v{:}\tau$. We conclude, taking $s' = s$, $\sigma' = \emptyset$ and $\mathcal{S}' = \mathcal{S}$, that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models v{:}\tau$$

Case of (abs) By hypothesis $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rules (abs) of the dynamic and static semantics impose that $s, E \vdash \mathtt{(lambda\ (x)\ e)} \rightarrow (\mathtt{x}, \mathtt{e}, E_{\mathtt{x}}), \emptyset, s$ and $\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset$. By the definition 3.2, taking $s' = s$, $\sigma' = \emptyset$ and $\mathcal{S}' = \mathcal{S}$, we conclude that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models (\mathtt{x}, \mathtt{e}, E_{\mathtt{x}}){:}\tau$$

Case of (let) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (let) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \mathtt{e}_1 \rightarrow v_1, f_1, s_1 \qquad s_1, E_{\mathtt{x}} + \{\mathtt{x} \mapsto v_1\} \vdash \mathtt{e}_2 \rightarrow v_2, f_2, s'}{s, E \vdash \mathtt{(let\ (x\ e_1)\ e_2)} \rightarrow v_2, f_1 \cup f_2, s'}$$

In the static semantics, writing $\sigma' = \sigma_1 \cup \sigma_2$, we have

$$\frac{\mathcal{E} \vdash \mathtt{e}_1 : \tau_1, \sigma_1 \qquad \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \mathit{Gen}(\sigma_1, \mathcal{E})(\tau_1)\} \vdash \mathtt{e}_2 : \tau_2, \sigma_2}{\mathcal{E} \vdash \mathtt{(let\ (x\ e_1)\ e_2)} : \tau_2, \sigma_1 \cup \sigma_2}$$

Let us write $\vec{v'} = \mathit{fv}(\tau_1) \setminus (\mathit{fv}(\sigma_1) \cup \mathit{fv}(\mathcal{E}))$. Let us define the substitution θ on $\vec{v'}$ such that the variables $\vec{v} = \theta(\vec{v'})$ are distinct and not free in $\sigma \cup \sigma_1$ and $\mathcal{E}$. Let us write $\tau = \theta\tau_1$. By the lemma 3.4,

$$\mathcal{E} \vdash \mathtt{e}_1 : \tau, \sigma_1$$

By induction hypothesis on $\mathtt{e}_1$, there exists $\mathcal{S}_1$ such that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \text{ and } s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v_1{:}\tau$$

Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1)$, by the definition 3.4, $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E{:}\mathcal{E}$.

Since $\vec{v} = \mathit{fv}(\tau) \setminus (\mathit{fv}(\sigma_1) \cup \mathit{fv}(\mathcal{E}))$, $\vec{v} \cap \mathit{fv}(\sigma_1) = \emptyset$. Let θ be any substitution defined on $\vec{v}$. Since $\vec{v} \cap \mathit{fv}(\sigma_1) = \emptyset$ and $\vec{v} \cap \mathit{fv}(\sigma) = \emptyset$, by the lemma 3.10, $s_1{:}\sigma \cup \sigma_1, \theta\mathcal{S}_1 \models v_1{:}\theta\tau$. Since $\vec{v} \cap \mathit{fv}(\sigma \cup \sigma_1) = \emptyset$, by the definition 3.1, $(\sigma \cup \sigma_1, \mathcal{S}_1) \simeq (\sigma \cup \sigma_1, \theta\mathcal{S}_1)$. By the lemma 3.8, $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v_1{:}\theta\tau$. By definition of $\preceq$ and definition 3.2,

$$s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v_1{:}\forall\vec{v}.\tau$$

Let us write $E' = E_{\mathtt{x}} + \{\mathtt{x} \mapsto v_1\}$ and $\mathcal{E}' = \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall\vec{v}.\tau\}$. By induction hypothesis on $\mathtt{e}_2$, there exists $\mathcal{S}'$ such that $(s_1, \sigma_1, \mathcal{S}_1) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$ and that $s'{:}\sigma \cup \sigma', \mathcal{S}' \models v_2{:}\tau_2$. Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1)$ and $(s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$, by the definition 3.4,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models v_2{:}\tau_2$$

Case of (app) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (app) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \mathtt{e}_1 \rightarrow (\mathtt{x}, \mathtt{e}_3, E'), f_1, s_1 \qquad s_1, E \vdash \mathtt{e}_2 \rightarrow v_2, f_2, s_2 \qquad s_2, E' + \{\mathtt{x} \mapsto v_2\} \vdash \mathtt{e}_3 \rightarrow v_3, f_3, s_3}{s, E \vdash \mathtt{(e_1\ e_2)} \rightarrow v_3, f_1 \cup f_2 \cup f_3, s_3}$$

Let us write $\sigma' = \sigma_1 \cup \sigma_2 \cup \sigma_3$. In the static semantics, we have:

$$\frac{\mathcal{E} \vdash \mathtt{e}_1 : \tau_2 \xrightarrow{\sigma_3} \tau_3, \sigma_1 \qquad \mathcal{E} \vdash \mathtt{e}_2 : \tau_2, \sigma_2}{\mathcal{E} \vdash \mathtt{(e_1\ e_2)} : \tau_3, \sigma_1 \cup \sigma_2 \cup \sigma_3}$$

By induction hypothesis on $\mathtt{e}_1$, there exists $\mathcal{S}_1$ such that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \text{ and } s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models (\mathtt{x}, \mathtt{e}_3, E'){:}\tau_2 \xrightarrow{\sigma_3} \tau_3$$

Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1)$, by the definition 3.4, $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E{:}\mathcal{E}$. By induction hypothesis on $\mathtt{e}_2$, there exists $\mathcal{S}_2$ such that

$$(s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \sqsubseteq (s_2, \sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \text{ and } s_2{:}\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2 \models v_2{:}\tau_2$$

Since $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models (\mathtt{x}, \mathtt{e}_3, E'){:}\tau_2 \xrightarrow{\sigma_3} \tau_3$ and by the definition 3.4, $s_2{:}\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2 \models (\mathtt{x}, \mathtt{e}_3, E'){:}\tau_2 \xrightarrow{\sigma_3} \tau_3$. By definition 3.2, this requires that there exist $\mathcal{E}'$, $\mathcal{S}'_2$ and σ'_2 such that $(\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \simeq (\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}'_2)$, $\mathit{Observe}(\tau_2 \xrightarrow{\sigma_3} \tau_3)(\sigma'_2) = \emptyset$ and $\mathit{Observe}(\sigma \cup \sigma_1 \cup \sigma_2)(\sigma'_2) = \emptyset$, verifying

$$\mathcal{E}' \vdash \mathtt{(lambda\ (x)\ e_3)} : \tau_2 \xrightarrow{\sigma_3} \tau_3, \emptyset \text{ and } s_2{:}\sigma \cup \sigma_1 \cup \sigma_2 \cup \sigma'_2, \mathcal{S}'_2 \models E'{:}\mathcal{E}'$$

Let us write $\vec{\rho} = \mathit{Regs}(\sigma'_2)$ and define θ on $\vec{\rho}$ by regions $\theta(\vec{\rho})$ not free in $\mathcal{E}$, $\sigma \cup \sigma_1 \cup \sigma_2$ and $\tau_2 \xrightarrow{\sigma_3} \tau_3$. $\mathit{Observe}(\tau_2 \xrightarrow{\sigma_3} \tau_3)(\sigma'_2) = \emptyset$ and $\mathit{Observe}(\sigma \cup \sigma_1 \cup \sigma_2)(\sigma'_2) = \emptyset$, by the lemma 3.2,

$$fv(\tau_2 \xrightarrow{\sigma_3} \tau_3) \cap Regs(\sigma_2') = \emptyset \text{ and } fv(\sigma \cup \sigma_1 \cup \sigma_2) \cap Regs(\sigma_2') = \emptyset$$

Let us write $\sigma_2'' = \theta\sigma_2'$ and $\mathcal{S}_2'' = \theta\mathcal{S}_2'$, we have $(\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \simeq (\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2'')$, $Observe(\tau_2 \xrightarrow{\sigma_3} \tau_3)(\sigma_2'') = \emptyset$ and $Observe(\sigma \cup \sigma_1 \cup \sigma_2)(\sigma_2'') = \emptyset$. By the lemmas 3.4 and 3.9, $\mathcal{E}'$, σ_2'' and $\mathcal{S}_2''$ verify

$$\theta\mathcal{E}' \vdash \texttt{(lambda (x) e}_3\texttt{)} : \tau_2 \xrightarrow{\sigma_3} \tau_3, \emptyset \text{ and } s_2{:}\sigma \cup \sigma_1 \cup \sigma_2 \cup \sigma_2'', \mathcal{S}_2'' \models E'{:}\theta\mathcal{E}'$$

Let us write $E'' = E'_{\mathtt{x}} + \{\mathtt{x} \mapsto v_2\}$ and $\mathcal{E}'' = \theta\mathcal{E}'_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau_2\}$. By the definition 3.2, $s_2{:}\sigma \cup \sigma_1 \cup \sigma_2 \cup \sigma_2'', \mathcal{S}_2'' \models E''{:}\mathcal{E}''$. By induction on $\mathtt{e}_3$, there exists $\mathcal{S}'$ such that

$$(s_2, \sigma \cup \sigma_1 \cup \sigma_2 \cup \sigma_2'', \mathcal{S}_2'') \sqsubseteq (s', \sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}') \text{ and that } s', \sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}' \models v_3{:}\tau_3$$

Since $s', \sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}' \models v_3{:}\tau_3$, $Observe(\tau_3)(\sigma_2') = \emptyset$ and $Observe(\sigma \cup \sigma')(\sigma_2') = \emptyset$, by the lemma 3.13,

$$s', \sigma \cup \sigma', \mathcal{S}' \models v_3{:}\tau_3$$

In the same manner, let us consider any v' and τ' such that $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$. Since $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}')$, by the definition 3.4, $s'{:}\sigma \cup \sigma' \cup \sigma_2'', \mathcal{S}' \models v'{:}\tau'$. Since, we can freely choose $\sigma_2'' = \theta\sigma_2'$ so that $Regs(\sigma_2'') \cap fr(\tau') = \emptyset$, by the definition of $Observe$, $Observe(\tau')(\sigma_2'') = \emptyset$. Thus, by the lemma 3.13, $s'{:}\sigma \cup \sigma', \mathcal{S}' \models v'{:}\tau'$. This holds for any judgment $s{:}\sigma, \mathcal{S} \models v'{:}\tau'$. Thus, by the definition 3.1,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$$

Case of (new) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (new) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \mathtt{e} \rightarrow v, f, s_1 \qquad l \notin Dom(s_1)}{s, E \vdash \texttt{(new e)} \rightarrow l, f \cup \{init(l)\}, s_1 + \{l \mapsto v\}}$$

In the static semantics, the situation is

$$\frac{\mathcal{E} \vdash \mathtt{e} : \tau, \sigma_1}{\mathcal{E} \vdash \texttt{(new e)} : ref_\rho(\tau), \sigma_1 \cup init(\rho, \tau)}$$

Let us write $\sigma' = init(\rho, \tau) \cup \sigma_1$. By induction hypothesis on $\mathtt{e}$, there exists $\mathcal{S}_1$ such that $(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1)$ and that $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v{:}\tau$. Let us write $(s', \sigma \cup \sigma', \mathcal{S}') = (s_1 + \{l \mapsto v\}, \sigma \cup \sigma', \mathcal{S}_1 + \{l \mapsto (\rho, \tau)\})$. Since $l \notin Dom(s_1)$ and $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models v{:}\tau$, by the lemma 3.11, $(s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$. Thus, by the definitions 3.4 and 3.3,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma', \mathcal{S}' \models l{:}ref_\rho(\tau)$$

Case of (get) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (get) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \mathtt{e} \rightarrow l, f_1, s' \qquad l \in Dom(s')}{s, E \vdash \texttt{(get e)} \rightarrow s'(l), f_1 \cup \{read(l)\}, s'}$$

In the static semantics, the rule (get) reads

$$\frac{\mathcal{E} \vdash \mathtt{e} : ref_\rho(\tau), \sigma_1}{\mathcal{E} \vdash \texttt{(get e)} : \tau, \sigma_1 \cup read(\rho, \tau)}$$

Let us write $\sigma' = read(\rho, \tau) \cup \sigma_1$. By induction hypothesis on $\mathtt{e}$, there exists $\mathcal{S}'$ such that $(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma_1, \mathcal{S}')$ and that $s'{:}\sigma \cup \sigma_1, \mathcal{S}' \models l{:}ref_\rho(\tau)$. By the definitions 3.4 and 3.2,

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma \cup \sigma', \mathcal{S}' \models s'(l){:}\tau$$

Case of (set) By hypothesis, $s{:}\sigma, \mathcal{S} \models E{:}\mathcal{E}$. The rule (set) of the dynamic semantics imposes that:

$$\frac{s, E \vdash \mathtt{e}_1 \rightarrow l, f_1, s_1 \qquad s_1, E \vdash \mathtt{e}_2 \rightarrow v, f_2, s_2}{s, E \vdash \mathtt{((set\ e_1)\ e_2)} \rightarrow u, f_1 \cup f_2 \cup \{write(l)\}, s_{2_l} + \{l \mapsto v\}}$$

In the static semantics, we have that

$$\frac{\mathcal{E} \vdash \mathtt{e}_1 : ref_\rho(\tau), \sigma_1 \qquad \mathcal{E} \vdash \mathtt{e}_2 : \tau, \sigma_2}{\mathcal{E} \vdash \mathtt{((set\ e_1)\ e_2)} : unit, \sigma_1 \cup \sigma_2 \cup write(\rho, \tau)}$$

Let us write $\sigma' = \sigma_1 \cup \sigma_2 \cup write(\rho, \tau)$. By induction hypothesis on $\mathtt{e}_1$, there exists $\mathcal{S}_1$ such that $(s, \sigma, \mathcal{S}) \sqsubseteq (s_1, \sigma \cup \sigma_1, \mathcal{S}_1)$ and that $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models l{:}ref_\rho(\tau)$. By the definition 3.4, $s_1{:}\sigma \cup \sigma_1, \mathcal{S}_1 \models E{:}\mathcal{E}$. By induction hypothesis on $\mathtt{e}_2$, there exists $\mathcal{S}_2$ such that

$$(s_1, \sigma \cup \sigma_1, \mathcal{S}_1) \sqsubseteq (s_2, \sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \text{ and that } s_2{:}\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2 \models v{:}\tau$$

By the definition 3.4, $s_2{:}\sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2 \models l{:}ref_\rho(\tau)$. Let us write $(s', \sigma \cup \sigma', \mathcal{S}') = (s_{2_l} + \{l \mapsto v\}, \sigma \cup \sigma', \mathcal{S}_2)$. By the lemma 3.12, $(s_2, \sigma \cup \sigma_1 \cup \sigma_2, \mathcal{S}_2) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}')$. By the definitions 3.4 and 3.2, we conclude that

$$(s, \sigma, \mathcal{S}) \sqsubseteq (s', \sigma \cup \sigma', \mathcal{S}') \text{ and } s'{:}\sigma', \mathcal{S}' \models u{:}unit \ \square$$

3.4 The Reconstruction Algorithm

We present the inference algorithm $\mathcal{I}$ that reconstructs the principal type and effect of expressions with respect to the static semantics. The inference algorithm $\mathcal{I}$ uses a double recursion scheme that separates the reconstruction of types and effects from the process of restricting effects with regard to the observation criterion.

In the first phase of the reconstruction, the algorithm $\mathcal{I}'$, given an environment $\mathcal{E}$ and a constraint set κ, reconstructs the type τ and the effect σ of an expression $\mathtt{e}$, together with a substitution θ that ranges over the free variables of the environment $\mathcal{E}$ and an updated constraint set κ'. In its second phase, the algorithm $\mathcal{I}$ takes into account the observation criterion *Observe* in order to restrict the effect σ computed by the algorithm $\mathcal{I}'$.

$$\mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}) = \text{let } (\theta, \tau, \sigma, \kappa') = \mathcal{I}'(\mathcal{E}, \kappa, \mathtt{e}) \text{ in } (\theta, \tau, Observe(\overline{\kappa}'(\theta\overline{\mathcal{E}}), \overline{\kappa}'\tau)(\overline{\kappa}'\sigma), \kappa')$$

$$\begin{aligned}
\mathcal{I}'(\mathcal{E},\kappa,\mathtt{e}) &= \text{case } \mathtt{e} \text{ of} \\
\mathtt{op} &\Rightarrow \text{let } (\tau', \kappa') = Inst(TypeOf[\![\mathtt{e}]\!]) \text{ in } (Id, \tau', \emptyset, \kappa \cup \kappa') \\
\mathtt{x} &\Rightarrow \text{if } \mathtt{x} \in Dom(\mathcal{E}) \text{ then let } (\tau', \kappa') = Inst(\mathcal{E}(\mathtt{x})) \text{ in } (Id, \tau', \emptyset, \kappa \cup \kappa') \text{ else fail} \\
\mathtt{(lambda\ (x)\ e)} &\Rightarrow \text{let } \alpha, \varsigma \text{ new and } (\theta, \tau, \sigma, \kappa') = \mathcal{I}(\mathcal{E}_\mathtt{x} + \{\mathtt{x} \mapsto \alpha\}, \kappa, \mathtt{e}) \\
&\quad \text{in } (\theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa' \cup \{\varsigma \supseteq \sigma\}) \\
\mathtt{(e\ e')} &\Rightarrow \text{let } (\theta, \tau, \sigma, \kappa') = \mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}) \\
&\quad\quad (\theta', \tau', \sigma', \kappa'') = \mathcal{I}(\theta\mathcal{E}, \kappa', \mathtt{e}') \\
&\quad\quad \alpha, \varsigma \text{ new and } \theta'' = \mathcal{U}_{\kappa''}(\theta'\tau, \tau' \xrightarrow{\varsigma} \alpha) \\
&\quad \text{in } (\theta'' \circ \theta' \circ \theta, \theta''\alpha, \theta''(\theta'\sigma \cup \sigma' \cup \varsigma), \theta''\kappa'') \\
\mathtt{(let\ (x\ e)\ e')} &\Rightarrow \text{let } (\theta, \tau, \sigma, \kappa') = \mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}) \text{ and } (\forall\vec{v}.(\tau, \kappa''), \kappa''') = Gen_{\kappa'}(\theta\mathcal{E}, \sigma)(\tau) \\
&\quad\quad (\theta', \tau', \sigma', \kappa'''') = \mathcal{I}(\theta\mathcal{E}_\mathtt{x} + \{\mathtt{x} \mapsto \forall\vec{v}.(\tau, \kappa'')\}, \kappa''', \mathtt{e}') \\
&\quad \text{in } (\theta' \circ \theta, \tau', \theta'\sigma \cup \sigma', \kappa'''')
\end{aligned}$$

Reconstruction Algorithm $\mathcal{I}$

3.4.1 Constrained Type Schemes

We use *constrained type schemes* to generically represent the possible types and constraint sets of `let`-bound expressions. In the static semantics, type schemes were of the form $\forall\vec{v}.\tau$. But now, since effect variables occur in function types, constraint sets involving these effect variables have to be kept within type schemes. In the algorithm, the type environment $\mathcal{E}$ binds value identifiers to such constrained type schemes.

Constrained type schemes, written $\forall\vec{v}.(\tau,\kappa)$ or $\forall\vec{\alpha}.\forall\vec{\varrho}.\forall\vec{\varsigma}.(\tau,\kappa)$, are composed of a type τ and a set of inequalities κ universally quantified over type, effect and region variables. The type and constraint set associated with `e` only depend on the free variables of `e` and, thereby, on the type environment $\mathcal{E}$. We write $\forall\vec{v}.(\tau,\emptyset) = \forall\vec{v}.\tau$

In order to relate the constrained type schemes and environments of the algorithm to the static semantics, we define a relation from the former to the latter by using the notion of principal model of constraint sets: $\overline{\forall\vec{v}.(\tau,\kappa)} = \forall\vec{v}.(\overline{\kappa}\tau)$ and $\overline{\mathcal{E}}(\mathtt{x}) = \overline{\mathcal{E}(\mathtt{x})}$ for all $\mathtt{x} \in Dom(\mathcal{E})$.

$$Gen_\kappa(\mathcal{E},\sigma)(\tau) = \text{let } \{\vec{v}\} = fv(\overline{\kappa}\tau) \setminus (fv(\overline{\kappa}\overline{\mathcal{E}}) \cup fv(\overline{\kappa}\sigma)) \text{ in } (\forall\vec{v}.(\tau,\kappa_{\vec{v}}), \kappa \setminus \kappa_{\vec{v}})$$
$$Inst(\forall\vec{v}.(\tau,\kappa)) = \text{let } \vec{v'} \text{ new and } \theta = \{\vec{v} \mapsto \vec{v'}\} \text{ in } (\theta\tau, \theta\kappa)$$

Generalization and Instantiation

For a given constraint set κ, the function Gen_κ generalizes the type τ of an expression upon the variables that are neither free in its environment $\mathcal{E}$ nor present in its observed effects σ. We write $\kappa_{\vec{v}}$ for the restriction $\kappa_{\vec{v}} = \{\varsigma \supseteq \sigma \in \kappa \mid \varsigma \in \vec{v}\}$ of κ on the effect variables $\vec{v}$. We write $\kappa \setminus \kappa_{\vec{v}}$ the complement of $\kappa_{\vec{v}}$ in κ. The instantiation of type schemes for value identifiers and operators is done by using the appropriate function *Inst*.

$$\begin{aligned}
TypeOf[\![\mathtt{set}]\!] &= \forall\alpha\varrho\varsigma\varsigma'.(ref_\varrho(\alpha) \xrightarrow{\varsigma} \alpha \xrightarrow{\varsigma'} unit, \{\varsigma' \supseteq write(\varrho,\alpha)\}) \\
TypeOf[\![\mathtt{get}]\!] &= \forall\alpha\varrho\varsigma.(ref_\varrho(\alpha) \xrightarrow{\varsigma} \alpha, \{\varsigma \supseteq read(\varrho,\alpha)\}) \\
TypeOf[\![\mathtt{new}]\!] &= \forall\alpha\varrho\varsigma.(\alpha \xrightarrow{\varsigma} ref_\varrho(\alpha), \{\varsigma \supseteq init(\varrho,\alpha)\})
\end{aligned}$$

Constrained Type Schemes for Store Operations

In the algorithm, the store operation `new`, `get` and `set` are best viewed as associated with appropriate constrained type schemes.

3.5 Constraint Resolution

We view the inference of types and effects of an expression as a constraint satisfaction problem. The algorithm builds equations on types and inequations on effects. In the algorithm, indirections between types and effects are introduced by the notion of constraint sets. Among the solutions of a constraint set, the principal model still satisfies the lemma of chapter 2: "If θ solves κ then $\theta = \theta \circ \overline{\kappa}$".

However, we introduced types in effects so that effects σ occur in types $\tau \xrightarrow{\sigma} \tau'$ as well as types τ occur in effects $init(\rho,\tau)$. As a consequence, some expressions may now have recursively defined types and effects and shall thus be rejected by the static semantics.

Example The static semantics might constraint some expressions to have an effect σ containing $init(\rho, \tau \xrightarrow{\sigma} \tau')$ itself. The simplest known example producing such an ill-formed constraint set is:

```
(lambda (f)
  (let (x (new (new (lambda (x) x))))
    (if true f (lambda (y) (set x (new f))) y)))
```

In this program, the type of the function `f` has to match the type of the lambda-expression `(lambda (y) (set x (new f)) y)` that initializes an observable reference to `f`. Note that the incriminated effect must be observable for this situation to appear. As a result, the latent effect of the function type for `f` is recursively defined.

$$\alpha \xrightarrow{\varsigma \cup write(\rho',\ldots) \cup init(\rho, \alpha \xrightarrow{\varsigma \cup \ldots} \alpha')} \alpha$$

For our algorithm to be effectively implemented constraint sets must be checked for well-formedness. It must be checked that no indirect cycles are introduced through *init* effects ■

3.5.1 Well-Formed Constraint Sets

Our solution consists of specifying well-formed constraint sets, which are the only acceptable to the algorithm $\mathcal{I}$, in that they correspond to sound assignments of effect variables in the static semantics.

Definition 3.5 (Well-Formed Constraint Set) *A constraint set κ is* well formed, *written $wf(\kappa)$, if and only if, for every $\varsigma \supseteq \sigma$ such that $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$ we have:*

$$\forall\ (\rho, \tau) \in Rng(\overline{\kappa}'\sigma),\ \varsigma \notin fv(\tau)$$

The notation $wf(\kappa)$ is extended to type schemes by $wf(\forall \vec{v}.(\tau, \kappa))$ iff $wf(\kappa)$ and type environments by $wf(\mathcal{E})$ iff $wf(\mathcal{E}(\mathtt{x}))$ for every $\mathtt{x}$ in $Dom(\mathcal{E})$.

The definition of well-formed constraint sets comes here with the following lemmas that state that well-formed constraint sets are solvable by finite substitutions.

Lemma 3.14 (Well-Formed Constraint Sets) *$wf(\kappa)$ if and only if $\overline{\kappa} \models \kappa$*

Proof We first prove that $wf(\kappa)$ implies $\overline{\kappa} \models \kappa$. The proof is by induction on the number of constraints in κ. If $\kappa = \emptyset$, then $\overline{\kappa} = Id$ solves κ. Consider $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$ where $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$. By definition, we have $\overline{\kappa} = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\} \circ \overline{\kappa}'$ and by induction hypothesis on κ', $\overline{\kappa}'$ solves κ'. For every constraint $\varsigma' \supseteq \sigma'$ in κ', $\overline{\kappa}(\varsigma') = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\varsigma'))$ and $\overline{\kappa}\sigma' = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\sigma'))$.

- If $\varsigma \in \overline{\kappa}'(\varsigma')$ then $\overline{\kappa}(\varsigma') = (\overline{\kappa}'(\varsigma') \setminus \varsigma) \cup \overline{\kappa}'(\varsigma \cup \sigma)$. Since $\varsigma \in \overline{\kappa}'(\varsigma)$ we have $\overline{\kappa}(\varsigma') = \overline{\kappa}'(\varsigma') \cup \overline{\kappa}'(\varsigma \cup \sigma)$. By induction we have $\overline{\kappa}'(\varsigma') \supseteq \overline{\kappa}'(\sigma')$. If $\varsigma \in \overline{\kappa}'(\sigma')$ then $\overline{\kappa}(\sigma') = (\overline{\kappa}'(\sigma') \setminus \varsigma) \cup \overline{\kappa}'(\varsigma \cup \sigma)$ so that $\overline{\kappa}(\varsigma') \supseteq \overline{\kappa}(\sigma')$. Otherwise $\varsigma \notin \overline{\kappa}'(\sigma')$ so $\overline{\kappa}(\sigma') = \overline{\kappa}'(\sigma')$; thus $\overline{\kappa}(\varsigma') = \overline{\kappa}'(\varsigma') \cup \overline{\kappa}'(\{\varsigma\} \cup \sigma) \supseteq \overline{\kappa}'(\varsigma') \supseteq \overline{\kappa}'(\sigma') = \overline{\kappa}(\sigma')$
- Otherwise $\varsigma \notin \overline{\kappa}'(\varsigma')$ and $\overline{\kappa}(\varsigma') = \overline{\kappa}'(\varsigma')$. Since $\varsigma \notin \overline{\kappa}'(\varsigma')$ and $\overline{\kappa}'(\varsigma') \supseteq \overline{\kappa}'(\sigma')$ we have $\varsigma \notin \overline{\kappa}'(\sigma')$, so $\overline{\kappa}(\sigma') = \overline{\kappa}'(\sigma')$. Since $\overline{\kappa}'$ solves κ', we have that $\overline{\kappa}'(\varsigma') \supseteq \overline{\kappa}'(\sigma')$ so that $\overline{\kappa}(\varsigma') \supseteq \overline{\kappa}(\sigma')$.

It follows that $\overline{\kappa}\varsigma' \supseteq \overline{\kappa}\sigma'$ in both cases. For every constraint $\varsigma' \supseteq \sigma'$ in κ'; so $\overline{\kappa}$ solves κ'.

It remains to show that $\overline{\kappa}$ solves $\{\varsigma \supseteq \sigma\}$. By definition $\overline{\kappa}(\varsigma) = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\varsigma))$. Since $\varsigma \in \overline{\kappa}'(\varsigma)$, $\overline{\kappa}(\varsigma) = (\overline{\kappa}'(\varsigma) \setminus \varsigma) \cup \overline{\kappa}'(\varsigma \cup \sigma) = \overline{\kappa}'(\varsigma) \cup \overline{\kappa}'(\sigma)$. Also, $\overline{\kappa}(\sigma) = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\sigma))$. If $\varsigma \in \overline{\kappa}'(\sigma)$, then $\overline{\kappa}(\sigma) = \overline{\kappa}'(\varsigma) \cup \overline{\kappa}'(\sigma)$, otherwise $\overline{\kappa}(\sigma) = \overline{\kappa}'(\sigma)$. In both cases, we have that $\overline{\kappa}$ solves $\{\varsigma \supseteq \sigma\}$. We have thus proved that $\overline{\kappa}$ solves κ.

Now, we prove that $\overline{\kappa} \models \kappa$ implies $wf(\kappa)$. Let us assume that $\overline{\kappa} \models \kappa$ and consider any constraint $\varsigma \supseteq \sigma$ in κ. Let us define $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$ where $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$ and $\overline{\kappa} = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\} \circ \overline{\kappa}'$. By hypothesis, we must have:

$$\{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\varsigma)) \supseteq \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\sigma))$$

Since $\varsigma \in \overline{\kappa}'\varsigma$, then $\{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\varsigma)) = \overline{\kappa}'(\varsigma \cup \sigma) = \overline{\kappa}'\varsigma \cup \overline{\kappa}'\sigma$. Thus, $\overline{\kappa}'\varsigma \cup \overline{\kappa}'\sigma \supseteq \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\sigma))$. Every (ρ, τ) in $Rng(\overline{\kappa}'\sigma)$ must also appear in $\{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'(\sigma))$. So, $\tau = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}\tau$. As a consequence, ς is not in $fv(\tau)$. This holds for every effect (ρ, τ) in $Rng(\overline{\kappa}'\sigma)$ and every constraint $\varsigma \supseteq \sigma$ in κ, so that we have $wf(\kappa)$ by definition □

We state that ill-formed constraint sets cannot be satisfied by substitutions of effect variables by finite effect terms.

Lemma 3.15 (Ill-Formed Constraint Sets) *If κ is ill-formed then there does not exist a substitution satisfying κ.*

Proof We show that in order to satisfy an ill-formed constraint set κ, any substitution θ must substitute at least one effect variable by a non finite effect term.

We assume that $\neg\, wf(\kappa)$. By definition, this implies that there exists a constraint $\{\varsigma \supseteq \sigma\}$ in κ such that $(\rho, \tau) \in Rng(\sigma)$, $\varsigma \in fv(\overline{\kappa}'\tau)$ where $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$. Suppose that there exists a substitution θ such that $\theta \models \kappa$.

By the lemma 2.10, we know that $\theta = \theta \circ \overline{\kappa}$. By definition of $\overline{\kappa}$, we know that $\overline{\kappa} = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\} \circ \overline{\kappa}'$. Then, from $\theta \models \kappa$ and by definition, θ must verify:

$$\theta(\overline{\kappa}\varsigma) = \theta(\overline{\kappa}'(\varsigma \cup \sigma)) \supseteq \theta(\overline{\kappa}(\sigma))$$

The substitution θ must verify $\theta(\overline{\kappa}'\tau) = \theta(\overline{\kappa}\tau)$. However,

$$\begin{array}{lll} \theta(\overline{\kappa}\tau) & = \theta(\{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\}(\overline{\kappa}'\tau)) & \text{by definition of } \overline{\kappa} \\ & = \theta(\{\varsigma \mapsto \varsigma \cup \overline{\kappa}'(\sigma)\}(\{\varsigma \mapsto \overline{\kappa}'\varsigma\}(\overline{\kappa}'\tau))) & \text{since } \varsigma \in \overline{\kappa}'\varsigma,\ \overline{\kappa}'\varsigma = \varsigma \cup \overline{\kappa}'\varsigma \\ & = \theta(\{\varsigma \mapsto \varsigma \cup \overline{\kappa}'\sigma\}(\overline{\kappa}'\tau)) & \text{since } \{\varsigma \mapsto \overline{\kappa}'\varsigma\} \circ \overline{\kappa}' = \overline{\kappa}' \end{array}$$

The term $\theta\varsigma$ must recursively verify $\theta\varsigma = \theta(\varsigma \cup \overline{\kappa}'\sigma)$ where $(\theta\rho, \theta \circ \overline{\kappa}'\tau) \in Rng(\theta \circ \overline{\kappa}'\sigma)$ and $\theta\varsigma$ occur in $\theta(\overline{\kappa}'\tau)$ since $\varsigma \in fv(\overline{\kappa}'\tau)$. Thus, the term $\theta\varsigma$ is not finite and the substitution θ satisfying κ cannot thus be defined with finite type and effect terms□

3.5.2 Unification Algorithm

In the reconstruction algorithm $\mathcal{I}$, instead of checking the well-formedness of the constructed constraint set after each expression is typechecked, we implement an extended occurrence check test, reporting the construction of ill-formed constraint at the point of unifying effect variables.

$$\begin{array}{rl} \mathcal{U}_\kappa(\tau, \tau') & = \text{case } (\tau, \tau') \text{ of} \\ (unit, unit) & \Rightarrow Id \\ (\alpha, \alpha') & \Rightarrow \{\alpha \mapsto \alpha'\} \\ (\alpha, \tau) | (\tau, \alpha) & \Rightarrow \text{if } \alpha \in fv(\overline{\kappa}\tau) \text{ then fail else } \{\alpha \mapsto \tau\} \\ (ref_\varrho(\tau), ref_{\varrho'}(\tau')) & \Rightarrow \text{let } \theta = \{\varrho \mapsto \varrho'\} \text{ in } \mathcal{U}_{\theta\kappa}(\theta\tau, \theta\tau') \circ \theta \\ (\tau_i \xrightarrow{\varsigma} \tau_f, \tau_i' \xrightarrow{\varsigma'} \tau_f') & \Rightarrow \text{let } \theta_i = \mathcal{U}_\kappa(\tau_i, \tau_i'),\ \theta_f = \mathcal{U}_{\theta_i\kappa}(\theta_i\tau_f, \theta_i\tau_f') \text{ and } \theta = \{\theta_f(\theta_i\varsigma) \mapsto \theta_f(\theta_i\varsigma')\} \circ \theta_f \circ \theta_i \\ & \quad \text{in if } wf(\theta\kappa) \text{ then } \theta \text{ else fail} \\ \text{otherwise} & \Rightarrow \text{fail} \end{array}$$

Unification Algorithm

Equations on the types that are built by the reconstruction algorithm are solved with the following unification algorithm $\mathcal{U}$. It either fails or returns a substitution θ standing for the most general unifier of the two given type terms τ and τ', also checking that the substitution θ preserves the well-formedness of the given constraint set κ. The following soundness and completeness lemma give the invariants of the unification algorithm $\mathcal{U}$.

Lemma 3.16 (Soundness of $\mathcal{U}$) *If κ be well-formed and $\mathcal{U}_\kappa(\tau, \tau') = \theta$ then $\theta\kappa$ is well-formed and $\theta\tau = \theta\tau'$.*

Proof The algorithm $\mathcal{U}$ unifies the terms of a free algebra and only departs from soundness proof of [Robinson, 1965] in the case that requires the well-formedness of the constraint set to be checked: the case of $\tau' = \alpha$. By hypothesis, κ is well formed and $\mathcal{U}_\kappa(\tau, \alpha) = \theta$. By definition of $\mathcal{U}_\kappa$, this requires that $\alpha \notin fv(\overline{\kappa}\tau)$ and $\theta = \{\alpha \mapsto \tau\}$. Thus, $\theta\alpha = \theta\tau$ and it remains to prove that $\theta\kappa$ is well formed.

By hypothesis, we have that κ is well-formed. By definition, this requires that for every constraint $\varsigma \supseteq \sigma$ in κ, considering $\kappa' = \kappa \setminus \{\varsigma \supseteq \sigma\}$, we have:

$$\forall\ (\rho, \tau') \in Rng(\overline{\kappa}'\sigma),\ \varsigma \notin fv(\tau')$$

We want to show that $\theta\kappa$ is well-formed. By definition, this requires to show that for every $\theta\varsigma \supseteq \theta\sigma$ in $\theta\kappa$, considering $\theta\kappa' = \theta\kappa \setminus \{\theta\varsigma \supseteq \theta\sigma\}$, we have:

$$\forall\ (\theta\rho, \theta\tau') \in Rng(\overline{\theta\kappa'}(\theta\sigma)),\ \theta\varsigma \notin fv(\theta\tau')$$

By definition, $\theta = \{\alpha \mapsto \tau\}$. Thus, it is sufficient to show that $\varsigma \notin fv(\theta\tau')$. If $\alpha \notin fv(\tau')$, then $\theta\tau' = \tau'$ so that we have $\varsigma \notin fv(\theta\tau')$. Otherwise, we have that $\alpha \in fv(\tau')$. Since $\varsigma \notin fv(\tau')$ and $\theta\alpha = \tau$, it remains to show that $\varsigma \notin fv(\tau)$.

We already know that ς is such that $\kappa = \kappa' \cup \{\varsigma \supseteq \sigma\}$ and that there exists $(\rho, \tau') \in Rng(\overline{\kappa}'\sigma)$ supposed to be such that $\alpha \in fv(\tau')$. By definition of $\overline{\kappa}$, this requires that $\alpha \in fv(\overline{\kappa}\varsigma)$. Then, supposing that $\varsigma \in fv(\tau)$ implies that $\alpha \in fv(\overline{\kappa}\tau)$, which contradicts the hypothesis $\alpha \notin fv(\overline{\kappa}\tau)$.

We have proved that for every $\varsigma \supseteq \theta\sigma$ in $\theta\kappa$, considering $\theta\kappa' = \theta\kappa \setminus \{\varsigma \supseteq \theta\sigma\}$, we have that $\varsigma \notin fv(\theta\tau')$ for every $(\rho, \theta\tau')$ in $Rng(\overline{\theta\kappa'}(\theta\sigma))$. By definition, this proves that $\theta\kappa$ is well-formed □

Lemma 3.17 (Completeness of $\mathcal{U}$) *Let κ be well-formed. Whenever $\theta'\tau = \theta'\tau'$ for a substitution θ' satisfying κ, then $\mathcal{U}_\kappa(\tau, \tau') = \theta$, $\theta\kappa$ is well-formed and there exists a substitution model θ'' satisfying $\theta\kappa$ such that $\theta' = \theta'' \circ \theta$.*

Proof By hypothesis, κ is well-formed and there exists a substitution θ' satisfying κ such that $\theta'\tau = \theta'\tau'$. The case analysis which differ from the completeness proof of [Robinson, 1965] are those which require the well-formedness of the constraint set to be checked: $\tau = \alpha$ or $(\tau, \tau') = (\tau_i \xrightarrow{\varsigma} \tau_f, \tau_i' \xrightarrow{\varsigma'} \tau_f')$.

In the case of $\tau = \alpha$, the hypothesis is that $\theta'\tau = \theta'\alpha$. By the lemma 2.10, we have that $\theta' = \theta' \circ \overline{\kappa}$, so that $\theta'(\overline{\kappa}\tau) = \theta'\alpha$. This implies that α is not in $fv(\overline{\kappa}\tau)$, so that we get that $\theta = \{\alpha \mapsto \tau\} = \mathcal{U}(\alpha, \tau)$ by definition. Consider θ'' defined by $\theta''\alpha = \alpha$ and $\theta''v = \theta'v$ otherwise. We have that $\theta' = \theta'' \circ \theta$ and that θ'' satisfies $\theta\kappa$ since θ' satisfies κ. By the lemma 3.15, this implies that $\theta\kappa$ is well-formed.

In the case of $(\tau, \tau') = (\tau_i \xrightarrow{\varsigma} \tau_f, \tau_i' \xrightarrow{\varsigma'} \tau_f')$, the hypothesis requires that $\theta'\tau_i = \theta'\tau_i'$, $\theta'\tau_f = \theta'\tau_f'$ and $\theta'\varsigma = \theta'\varsigma'$.

By induction hypothesis on τ_i and τ_i', $\mathcal{U}_\kappa(\tau_i, \tau_i') = \theta_i$, $wf(\theta_i\kappa)$ and $\theta' = \theta_i'' \circ \theta_i$ for some $\theta_i'' \models \theta_i\kappa$. Since $wf(\theta_i\kappa)$ and since there exists a substitution θ_i'' satisfying $\theta_i\kappa$ such that $\theta_i''(\theta_i\tau) = \theta_i''(\theta_i\tau')$, then, by induction hypothesis on τ_f and τ_f', we have $\mathcal{U}_{\theta_i\kappa}(\theta_i\tau_f, \theta_i\tau_f') = \theta_f$, $wf(\theta_f(\theta_i\kappa))$ and $\theta_i'' = \theta_f'' \circ \theta_f$ for some $\theta_f'' \models \theta_f(\theta_i\kappa)$.

Thus, there exists a substitution θ_f'' satisfying $\theta_f(\theta_i\kappa)$ such that $\theta_f''(\theta_f(\theta_i\tau)) = \theta_f''(\theta_f(\theta_i\tau'))$. But $\theta_f''(\theta_f(\theta_i\tau)) = \theta_f''(\theta_f(\theta_i\tau'))$ requires that $\theta_f''(\theta_f(\theta_i\varsigma)) = \theta_f''(\theta_f(\theta_i\varsigma'))$. Let us write $\theta'' = \theta_f''$ and $\theta = \{\theta_f(\theta_i\varsigma) \mapsto \theta_f(\theta_i\varsigma')\} \circ \theta_f \circ \theta_i$.

We have $\theta_f'' \circ \theta_f \circ \theta_i = \theta'' \circ \theta$. Since the substitution θ_f'' satisfies $\theta_f(\theta_i\kappa)$ then $\theta'' \circ \theta$ satisfies κ so that θ'' satisfies $\theta\kappa$. By the lemma 3.15, this implies that $\theta\kappa$ is well formed. As a conclusion, we get $\mathcal{U}_\kappa(\tau, \tau') = \theta$, $wf(\theta\kappa)$ and the substitution θ'' satisfying $\theta\kappa$ such that $\theta' = \theta'' \circ \theta$ □

3.6 Correctness of the Reconstruction Algorithm

In this section, we prove the correctness of the algorithm with respect to the static semantics. The soundness theorem states that the type and effect computed by $\mathcal{I}$ are provable in the static semantics, assuming any solution of the inferred constraints.

Theorem 3.2 (Soundness) *Let $\mathcal{E}$ and κ be well-formed. If $\mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}) = (\theta, \tau, \sigma, \kappa')$ then $\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \mathtt{e} : \overline{\kappa}'\tau, \overline{\kappa}'\sigma$*

Proof The proof is by induction on the structure of expressions. We assume that the constrained type schemes in the environment $\mathcal{E}$ are constructed with the function *Gen*, so that, for every $\mathtt{x}$ in $Dom(\mathcal{E})$, we have that $\mathcal{E}(\mathtt{x}) = \forall\vec{v}.(\tau, \kappa)$ with κ restricted to the $\vec{v}$. Then, a consequence is that for any substitution θ, we have that $\theta\overline{\mathcal{E}}(\mathtt{x}) = \overline{\theta\mathcal{E}}(\mathtt{x})$, ignoring capture of bound variables. Also note that, by definition of the reconstruction algorithm, the constraint set κ' extends $\theta\kappa$; every model of κ' is thus a model of $\theta\kappa$.

Case of (var) By hypothesis $\mathcal{I}(\mathcal{E},\kappa,\mathtt{x}) = (Id,\theta\tau,\emptyset,\kappa\cup\theta\kappa')$. By the definition of the algorithm, this requires that $\mathcal{E}(\mathtt{x}) = \forall\vec{v}.(\tau,\kappa')$, that $\theta = \{\vec{v}\mapsto\vec{v'}\}$ and that the $\vec{v'}$ are fresh. Since θ renames the $\vec{v}$ with fresh $\vec{v'}$, we have that $\theta(\overline{\kappa'}\tau) = \overline{\theta\kappa'}(\theta\tau)$. Since $\overline{\kappa'}$ and θ are only defined on $\vec{v}$, we have $\overline{\theta\kappa'}(\overline{\mathcal{E}}) = \overline{\mathcal{E}}$. By definition of $\preceq$, $\theta(\overline{\kappa'}\tau)\preceq\overline{\mathcal{E}}(\mathtt{x})$. By definition of the rule (var):

$$\overline{\theta\kappa'}\ \overline{\mathcal{E}}\vdash\mathtt{x}:\overline{\theta\kappa'}(\theta\tau),\emptyset$$

Because θ substitutes $\vec{v}$ with fresh $\vec{v'}$, we have that $\overline{\kappa\cup\theta\kappa'} = \overline{\kappa}\circ\overline{\theta\kappa'}$. By the lemma 3.4 used with $\overline{\kappa}$, we can conclude that:

$$\overline{\kappa}(\overline{\theta\kappa'}\ \overline{\mathcal{E}})\vdash\mathtt{x}:\overline{\kappa}(\overline{\theta\kappa'}(\theta\tau)),\emptyset$$

Case of (let) By hypothesis $\mathcal{I}(\mathcal{E},\kappa,(\mathtt{let\ (x\ e_1)\ e_2})) = (\theta,\tau,\sigma,\kappa')$. By the definition of the algorithm $\mathcal{I}$, this requires that:

$$\mathcal{I}'(\mathcal{E},\kappa,,(\mathtt{let\ (x\ e_1)\ e_2})) = (\theta,\tau,\sigma',\kappa')\quad\text{and}\quad\sigma = Observe(\overline{\kappa}'(\theta\overline{\mathcal{E}}),\overline{\kappa}'\tau)(\overline{\kappa}'\sigma')$$

By the definition of $\mathcal{I}'$, this requires that there exist θ_1, θ_2 such that $\theta=\theta_2\circ\theta_1$ and σ_1, σ_2 such that $\sigma' = \theta_2\sigma_1\cup\sigma_2$ satisfying:

$$(\theta_1,\tau_1,\sigma_1,\kappa_1) = \mathcal{I}(\mathcal{E},\kappa,\mathtt{e}_1)\quad\text{and}\quad(\theta_2,\tau,\sigma_2,\kappa') = \mathcal{I}(\theta_1\mathcal{E}_{\mathtt{x}}+\{\mathtt{x}\mapsto\forall\vec{v}.(\tau_1,\kappa_1')\},\kappa_1'',\mathtt{e}_2)$$

where $Gen_{\kappa_1}(\sigma_1,\theta_1\mathcal{E})(\tau_1) = (\forall\vec{v}.(\tau_1,\kappa_1'),\kappa_1'')$. By induction hypothesis on $\mathtt{e}_1$, we get:

$$\overline{\kappa}_1(\theta_1\overline{\mathcal{E}})\vdash\mathtt{e}_1:\overline{\kappa}_1\tau_1,\overline{\kappa}_1\sigma_1$$

By the definition of *Gen*, we have that $\overline{\kappa}_1\tau_1 = \overline{\kappa}_1''(\overline{\kappa}_1'\tau_1)$, because κ_1' is the restriction of κ_1 on $\vec{v}$ and κ_1'' its complement in κ_1. We also have that $\overline{\kappa}_1(\theta_1\overline{\mathcal{E}}) = \overline{\kappa}_1''(\theta_1\overline{\mathcal{E}})$ and that $\overline{\kappa}_1\sigma_1 = \overline{\kappa}_1''\sigma_1$ since the $\vec{v}$ are neither free in $\overline{\kappa}_1(\overline{\theta_1\mathcal{E}})$ nor in $\overline{\kappa}_1\sigma_1$. Thus, we have that:

$$\overline{\kappa}_1''(\theta_1\overline{\mathcal{E}})\vdash\mathtt{e}_1:\overline{\kappa}_1''(\overline{\kappa}_1'\tau_1),\overline{\kappa}_1''\sigma_1$$

Since κ' extends $\theta_2\kappa_1''$, we have that $\overline{\kappa}'$ satisfies $\theta_2\kappa_1''$, so that $\overline{\kappa}'\circ\theta_2$ satisfies κ_1'' by definition. Then, by the lemma 2.10, we have that $\overline{\kappa}'\circ\theta_2 = \overline{\kappa}'\circ\theta_2\circ\overline{\kappa}_1''$. By the lemma 3.4 used with $\overline{\kappa}'\circ\theta_2$,

$$\overline{\kappa}'(\theta_2(\theta_1\overline{\mathcal{E}}))\vdash\mathtt{e}_1:\overline{\kappa}'(\theta_2(\overline{\kappa}_1'\tau_1)),\overline{\kappa}'(\theta_2\sigma_1)$$

By induction hypothesis on $\mathtt{e}_2$, with κ_1'' and $\theta_1\mathcal{E}_{\mathtt{x}}+\{\mathtt{x}\mapsto\forall\vec{v}.(\tau_1,\kappa_1')\}$, we get:

$$\overline{\kappa}'(\theta_2(\overline{\theta_1\mathcal{E}}_{\mathtt{x}}+\{\mathtt{x}\mapsto\overline{\forall\vec{v}.(\tau_1,\kappa_1')}\}))\vdash\mathtt{e}_2:\overline{\kappa}'\tau,\overline{\kappa}'\sigma_2$$

Since $\overline{\theta_1\mathcal{E}_{\mathtt{x}}} = \theta_1\overline{\mathcal{E}_{\mathtt{x}}}$, then $\overline{\theta_1\mathcal{E}}_{\mathtt{x}}+\{\mathtt{x}\mapsto\overline{\forall\vec{v}.(\tau_1,\kappa_1')}\} = \theta_1\overline{\mathcal{E}_{\mathtt{x}}}+\{\mathtt{x}\mapsto\forall\vec{v}.(\overline{\kappa}_1'\tau_1)\}$. Thus,

$$\overline{\kappa}'(\theta_2(\theta_1\overline{\mathcal{E}}_{\mathtt{x}}+\{\mathtt{x}\mapsto\forall\vec{v}.\overline{\kappa}_1'\tau_1\}))\vdash\mathtt{e}_2:\overline{\kappa}'\tau,\overline{\kappa}'\sigma_2$$

Since $\theta=\theta_2\circ\theta_1$ and by the definition of the rule (let), this implies that:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}})\vdash(\mathtt{let\ (x\ e_1)\ e_2}):\overline{\kappa}'\tau,\overline{\kappa}'(\theta_2\sigma_1\cup\sigma_2)$$

We know that $Observe(\overline{\kappa}'(\theta\overline{\mathcal{E}}),\overline{\kappa}'\tau)(\overline{\kappa}'\sigma') = \sigma$, so that, using the rule (sub), we get:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}})\vdash(\mathtt{let\ (x\ e_1)\ e_2}):\overline{\kappa}'\tau,\sigma$$

By the lemma 3.4 used with $\overline{\kappa}'$ and since $\overline{\kappa}'\circ\overline{\kappa}' = \overline{\kappa}'$, we conclude that:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}})\vdash(\mathtt{let\ (x\ e_1)\ e_2}):\overline{\kappa}'\tau,\overline{\kappa}'\sigma$$

Case of (abs) By hypothesis $\mathcal{I}(\mathcal{E},\kappa,\texttt{(lambda (x) e)}) = (\theta,\theta\alpha \xrightarrow{\varsigma} \tau,\emptyset,\kappa' \cup \{\varsigma \supseteq \sigma\})$. By the definition of the algorithm $\mathcal{I}$, this requires that $\mathcal{I}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\},\kappa,\mathtt{e}) = (\theta,\tau,\sigma,\kappa')$. By induction hypothesis on $\mathtt{e}$,

$$\overline{\kappa}'(\theta(\overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\})) \vdash \mathtt{e} : \overline{\kappa}'\tau, \overline{\kappa}'\sigma$$

Since ς is fresh, $\overline{\kappa}'(\sigma \cup \varsigma) = \overline{\kappa}'\sigma \cup \varsigma \supseteq \overline{\kappa}'\sigma$. Thus, by the rule (sub):

$$\overline{\kappa}'(\theta(\overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\})) \vdash \mathtt{e} : \overline{\kappa}'\tau, \overline{\kappa}'(\varsigma \cup \sigma)$$

Let us write $\kappa'' = \kappa' \cup \{\varsigma \supseteq \sigma\}$. By definition $\overline{\kappa}'' = \{\varsigma \mapsto \overline{\kappa}'(\varsigma \cup \sigma)\} \circ \overline{\kappa}'$. Since ς is fresh, we get:

$$\overline{\kappa}''(\theta(\overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\})) \vdash \mathtt{e} : \overline{\kappa}''\tau, \overline{\kappa}''\varsigma$$

By the rule (abs), we conclude that:

$$\overline{\kappa}''(\theta\overline{\mathcal{E}}) \vdash \texttt{(lambda (x) e)} : \overline{\kappa}''(\theta\alpha \xrightarrow{\varsigma} \tau), \emptyset$$

Case of (app) By hypothesis $\mathcal{I}(\mathcal{E},\kappa,(\mathtt{e}_1\ \mathtt{e}_2)) = (\theta,\theta_3\alpha,\sigma,\kappa')$. By definition of the algorithm $\mathcal{I}$, this requires that

$$\mathcal{I}(\mathcal{E},\kappa,\mathtt{e}_1) = (\theta_1,\tau_1,\sigma_1,\kappa_1) \qquad \mathcal{I}(\theta_1\mathcal{E},\kappa_1,\mathtt{e}_2) = (\theta_2,\tau_2,\sigma_2,\kappa_2) \quad \text{and} \quad \theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha)$$

where $\sigma = \mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}),\overline{\kappa}'(\theta_3\alpha))(\overline{\kappa}'\sigma_3)$, $\sigma_3 = \theta_3(\theta_2\sigma_1 \cup \sigma_2 \cup \varsigma)$ and $\kappa' = \theta_3\kappa_2$. By induction hypothesis on $\mathtt{e}_1$, we get

$$\overline{\kappa}_1(\theta_1\overline{\mathcal{E}}) \vdash \mathtt{e}_1 : \overline{\kappa}_1\tau_1, \overline{\kappa}_1\sigma_1$$

By definition of κ', $\overline{\kappa}'$ satisfies $\theta_3 \circ \theta_2\kappa_1$. Thus $\overline{\kappa}' \circ \theta_3 \circ \theta_2$ satisfies κ_1. By the lemma 2.10 on κ_1, we have $\overline{\kappa}' \circ \theta_3 \circ \theta_2 = \overline{\kappa}' \circ \theta_3 \circ \theta_2 \circ \overline{\kappa}_1$. By the lemma 3.4 used with $\overline{\kappa}' \circ \theta_3 \circ \theta_2$, we get:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \mathtt{e}_1 : \overline{\kappa}'(\theta_3(\theta_2\tau_1)), \overline{\kappa}'(\theta_3(\theta_2\sigma_1))$$

Since $\theta_3 = \mathcal{U}_{\kappa_2}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha)$ then $\theta_3\kappa_2$ is well-formed and $\theta_3(\theta_2\tau_1) = \theta_3(\tau_2 \xrightarrow{\varsigma} \alpha)$ by the correctness lemma on unification, yielding:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \mathtt{e}_1 : \overline{\kappa}'(\theta_3\tau_2) \xrightarrow{\overline{\kappa}'(\theta_3\varsigma)} \overline{\kappa}'(\theta_3\alpha), \overline{\kappa}'(\theta_3(\theta_2\sigma_1))$$

Since $\overline{\theta_1\mathcal{E}} = \theta_1\overline{\mathcal{E}}$ and by induction hypothesis on $\mathtt{e}_2$ with $\theta_1\mathcal{E}$,

$$\overline{\kappa}_2(\theta_2(\theta_1\overline{\mathcal{E}})) \vdash \mathtt{e}_2 : \overline{\kappa}_2\tau_2, \overline{\kappa}_2\sigma_2$$

Since $\overline{\kappa}'$ satisfies $\theta_3\kappa_2$, $\overline{\kappa}' \circ \theta_3$ satisfies κ_2. By the lemma 2.10 using $\overline{\kappa}_2$, we have $\overline{\kappa}' \circ \theta_3 = \overline{\kappa}' \circ \theta_3 \circ \overline{\kappa}_2$. By lemma 3.4 used with $\overline{\kappa}' \circ \theta_3$, we get:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash \mathtt{e}_2 : \overline{\kappa}'(\theta_3\tau_2), \overline{\kappa}'(\theta_3\sigma_2)$$

By definition of σ_3 and the rule (app),

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash (\mathtt{e}_1\ \mathtt{e}_2) : \overline{\kappa}'(\theta_3\alpha), \overline{\kappa}'\sigma_3$$

Since $\sigma = \mathit{Observe}(\overline{\kappa}'(\theta\overline{\mathcal{E}}),\overline{\kappa}'(\theta_3\alpha))(\overline{\kappa}'\sigma_3)$ and by the rule (sub),

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash (\mathtt{e}_1\ \mathtt{e}_2) : \overline{\kappa}'(\theta_3\alpha), \sigma$$

By the lemma 3.4 used with $\overline{\kappa}'$, we get:

$$\overline{\kappa}'(\theta\overline{\mathcal{E}}) \vdash (\mathtt{e}_1\ \mathtt{e}_2) : \overline{\kappa}'(\theta_3\alpha), \overline{\kappa}'\sigma \ \square$$

The completeness theorem states that the inferred type is principal with respect to substitutions on variables and that the reconstructed effect is minimal with respect to the subsumption on effects.

Theorem 3.3 (Completeness) *Let $\mathcal{E}$ and κ be well-formed and θ'' be a model of κ. If $\theta''\overline{\mathcal{E}} \vdash \mathtt{e} : \tau', \sigma'$ then $\mathcal{I}(\mathcal{E},\kappa,\mathtt{e}) = (\theta,\tau,\sigma,\kappa')$ and there exists $\theta' \models \kappa'$ such that $\theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}})$, $\tau' = \theta'\tau$ and $\sigma' \supseteq \theta'\sigma$*

Proof The proof is by induction on the structure of expressions.

Case of (var) By hypothesis, $\theta''\overline{\mathcal{E}} \vdash \mathtt{x} : \tau', \sigma'$. This requires that $\theta''\overline{\mathcal{E}}(\mathtt{x}) \succ \tau'$ by definition of the rule (var) and that $\mathcal{E}(\mathtt{x}) = \forall\vec{v}.(\tau, \kappa')$ by definition of $\overline{\mathcal{E}}$. By definition of the algorithm $\mathcal{I}'$, we get:

$$\mathcal{I}(\mathcal{E}, \kappa, \mathtt{x}) = (Id, \theta\tau, \emptyset, \kappa \cup \theta\kappa') \quad \text{and} \quad \theta = \{\vec{v} \mapsto \vec{v'}\}$$

By hypothesis, we have that $\tau' \preceq \theta''\overline{\mathcal{E}}(\mathtt{x})$. By definition of $\preceq$, this requires that there exists a substitution θ'_1 defined on $\vec{v}$ such that:

$$\theta'_1(\theta''(\overline{\kappa}'\tau)) = \tau'$$

By definition of the algorithm $\mathcal{I}'$, θ substitutes $\vec{v}$ with fresh $\vec{v'}$. Let θ'_2 be defined on $\vec{v'}$ by $\theta'_2(\vec{v'}) = \theta'_2(\theta(\vec{v})) = \theta'_1(\vec{v'})$. Avoiding capture, the substitution θ'_2 satisfies:

$$\theta'_2(\theta''(\theta(\overline{\kappa}'\tau))) = \tau'$$

Since θ renames $\vec{v}$ with fresh $\vec{v'}$, we have $\theta \circ \overline{\kappa}' = \overline{\theta\kappa'} \circ \theta$. As a consequence, $\theta' = \theta'_2 \circ \theta'' \circ \overline{\theta\kappa'}$ satisfies $\kappa \cup \theta\kappa'$ and is such that:

$$\tau' = \theta'(\theta\tau) \quad \text{and} \quad \theta''\overline{\mathcal{E}} = \theta'\overline{\mathcal{E}} \quad \text{and} \quad \sigma' \supseteq \emptyset$$

Case of (let) By hypothesis, $\theta''\overline{\mathcal{E}} \vdash \mathtt{(let\ (x\ e_1)\ e_2)} : \tau', \sigma'_1 \cup \sigma'_2$. By the definition of rule (let), this requires that

$$\theta''\overline{\mathcal{E}} \vdash \mathtt{e}_1 : \tau'_1, \sigma'_1 \quad \text{and} \quad \theta''\overline{\mathcal{E}_{\mathtt{x}}} + \{\mathtt{x} \mapsto Gen(\sigma'_1, \theta''\overline{\mathcal{E}})(\tau'_1)\} \vdash \mathtt{e}_2 : \tau', \sigma'_2$$

By induction hypothesis on $\mathtt{e}_1$, $\mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}_1) = (\theta_1, \tau_1, \sigma_1, \kappa_1)$ and there exists a substitution θ'_1 satisfying κ_1 such that:

$$\theta''\overline{\mathcal{E}} = \theta'_1(\theta_1\overline{\mathcal{E}}) \qquad \tau'_1 = \theta'_1\tau_1 \quad \text{and} \quad \sigma'_1 \supseteq \theta'_1\sigma_1$$

Since $\theta''\overline{\mathcal{E}} = \theta'_1(\theta_1\overline{\mathcal{E}})$, $\tau'_1 = \theta'_1\tau_1$, $\sigma'_1 \supseteq \theta'_1\sigma_1$ and by By definition of Gen,

$$\forall\tau, \quad \tau \preceq Gen(\sigma'_1, \theta''\overline{\mathcal{E}})(\tau'_1) \;\Rightarrow\; \tau \preceq Gen(\theta'_1\sigma_1, \theta'_1(\theta_1\overline{\mathcal{E}}))(\theta'_1\tau_1)$$

As a consequence, given that $\theta''\overline{\mathcal{E}_{\mathtt{x}}} = \theta'_1(\theta_1\overline{\mathcal{E}_{\mathtt{x}}})$ and by the lemma 3.5,

$$\theta'_1(\theta_1\overline{\mathcal{E}_{\mathtt{x}}}) + \{\mathtt{x} \mapsto Gen(\theta'_1\sigma_1, \theta'_1(\theta_1\overline{\mathcal{E}}))(\theta'_1\tau_1)\} \vdash \mathtt{e}_2 : \tau', \sigma'_2$$

Let $(\forall\vec{v}.(\tau_1, \kappa'_1), \kappa''_1) = Gen_{\kappa_1}(\sigma_1, \theta_1\mathcal{E})(\tau_1)$ and define θ''_1 by Id on $\vec{v}$ and by θ'_1 elsewhere. By definition of θ''_1,

$$\theta''_1(\theta_1\overline{\mathcal{E}}) = \theta'_1(\theta_1\overline{\mathcal{E}}) \quad \text{and} \quad \theta''_1\sigma_1 = \theta'_1\sigma_1$$

Let us consider any τ such that $\tau \preceq Gen(\theta'_1\sigma_1, \theta'_1(\theta_1\overline{\mathcal{E}}))(\theta'_1\tau_1)$. By definition of $\preceq$, there exists a substitution θ defined on $fv(\theta'_1\tau_1) \setminus (fv(\theta'_1\sigma_1) \cup fv(\theta'_1(\theta_1\overline{\mathcal{E}})))$ such that

$$\tau = \theta(\theta'_1\tau_1)$$

Let us define θ' by $\theta \circ \theta'_1$ on $\vec{v}$. Since θ is defined on $fv(\theta'_1\tau_1) \setminus (fv(\theta'_1\sigma_1) \cup fv(\theta'_1(\theta_1\overline{\mathcal{E}})))$, any variable v' in $fv(\theta'_1 v)$ satisfies $\theta v' = v'$. Thus

$$\theta \circ \theta'_1 = \theta' \circ \theta''_1$$

By the lemma 2.10, $\theta'_1 = \theta'_1 \circ \overline{\kappa}_1 = \theta'_1 \circ \overline{\kappa}'_1$. Thus,

$$\tau = \theta(\theta'_1\tau_1) = \theta(\theta'_1(\overline{\kappa}'_1\tau_1)) = \theta'(\theta''_1(\overline{\kappa}'_1\tau_1))$$

By definition of $\preceq$, this implies that:

$$\tau \preceq \theta''_1(\forall\vec{v}.\overline{\kappa}'_1\tau_1)$$

Thus, by the lemma 3.5, there exists a proof of there exists a proof of:

$$\theta''_1\overline{\theta_1\mathcal{E}_{\mathtt{x}}} + \{\mathtt{x} \mapsto \overline{\forall\vec{v}.(\tau_1, \kappa'_1)}\} \vdash \mathtt{e}_2 : \tau', \sigma'_2$$

Since $\overline{\theta_1 \mathcal{E}_{\mathtt{x}}} = \theta_1 \overline{\mathcal{E}_{\mathtt{x}}}$, since θ_1' satisfies κ_1'' and by induction hypothesis on $\mathtt{e}_2$ with κ_1'' and $\theta_1 \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall \vec{v}.(\tau_1, \kappa_1')\}$,

$$\mathcal{I}(\theta_1 \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall \vec{v}.(\tau_1, \kappa_1')\}, \kappa_1'', \mathtt{e}_2) = (\theta_2, \tau, \sigma_2, \kappa')$$

and there exists a substitution θ_2' satisfying κ' such that $\tau' = \theta_2' \tau$, $\sigma_2' \supseteq \theta_2' \sigma_2$ and

$$\theta_1'' \overline{\theta_1 \mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \overline{\forall \vec{v}.(\tau_1, \kappa_1')}\} = \theta_2'(\theta_2 \overline{\theta_1 \mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \overline{\forall \vec{v}.(\tau_1, \kappa_1')}\}) = \theta_2'(\theta_2 \theta_1 \overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall \vec{v}.(\overline{\kappa}_1' \tau_1)\})$$

Let us write $\theta = \theta_2 \circ \theta_1$, $\sigma_3 = \theta_2 \sigma_1 \cup \sigma_2$ and $\sigma = \mathit{Observe}(\overline{\kappa}'(\theta \overline{\mathcal{E}}), \overline{\kappa}' \tau)(\overline{\kappa}' \sigma_3)$. By definition of the algorithm, we get that:

$$\mathcal{I}(\mathcal{E}, \kappa, (\mathtt{let\ (x\ e_1)\ e_2})) = (\theta, \tau, \sigma, \kappa')$$

Let V be the free variables of $\overline{\kappa}' \theta \overline{\mathcal{E}}$, $\overline{\kappa}' \tau_2$ and $\overline{\kappa}' \sigma_2$. Define θ' by θ_2' on V and θ_1'' otherwise. By definition, θ' satisfies κ' and satisfies

$$\tau' = \theta' \tau \qquad \theta'' \overline{\mathcal{E}} = \theta'(\theta \overline{\mathcal{E}}) \quad \text{and} \quad \sigma_1' \cup \sigma_2' \supseteq \theta' \sigma_3$$

Since $\theta' \sigma = \theta' \mathit{Observe}(\overline{\kappa}'(\theta \overline{\mathcal{E}}), \overline{\kappa}' \tau)(\overline{\kappa}' \sigma_3)$ and $\theta' = \theta' \circ \overline{\kappa}'$, then, by the lemma 3.3,

$$\theta' \mathit{Observe}(\overline{\kappa}'(\theta \overline{\mathcal{E}}), \overline{\kappa}' \tau)(\overline{\kappa}' \sigma_3) \subseteq \mathit{Observe}(\theta'(\theta \overline{\mathcal{E}}), \theta' \tau)(\theta' \sigma_3)$$

We conclude that $\sigma_1' \cup \sigma_2' \supseteq \theta' \sigma$. We have proved that $\mathcal{I}(\mathcal{E}, \kappa, (\mathtt{let\ (x\ e_1)\ e_2})) = (\theta, \tau, \sigma, \kappa')$ and that there exists a substitution θ' satisfying κ' such that:

$$\theta'' \overline{\mathcal{E}} = \theta'(\theta \overline{\mathcal{E}}),\ \tau' = \theta' \tau \ \mathit{and}\ \sigma_1' \cup \sigma_2' \supseteq \theta' \sigma$$

Case of (abs) By hypothesis, $\theta'' \overline{\mathcal{E}} \vdash (\mathtt{lambda\ (x)\ e}) : \tau_i \xrightarrow{\sigma'} \tau_f, \sigma''$. By definition of the rule (abs), this requires that

$$\theta'' \overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau_i\} \vdash \mathtt{e} : \tau_f, \sigma'$$

With a fresh variable α, this is equivalent to:

$$(\theta'' \circ \{\alpha \mapsto \tau_i\}) \overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\} \vdash \mathtt{e} : \tau_f, \sigma'$$

By induction hypothesis on $\mathtt{e}$ since $\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}$ is well-formed, we have that:

$$\mathcal{I}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}, \kappa, \mathtt{e}) = (\theta, \tau, \sigma, \kappa')$$

and there exists a substitution θ_1' satisfying κ' such that:

$$(\theta'' \circ \{\alpha \mapsto \tau_i\}) \overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\} = \theta_1'(\theta \overline{\mathcal{E}}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}),\ \tau_f = \theta_1' \tau \quad \text{and} \quad \sigma' \supseteq \theta_1' \sigma$$

By definition of the algorithm, we get that:

$$\mathcal{I}(\mathcal{E}, \kappa, (\mathtt{lambda\ (x)\ e})) = (\theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa' \cup \{\varsigma \supseteq \sigma\})$$

where ς is new. Let us consider the model $\theta' = \theta_1' \circ \{\varsigma \mapsto \sigma'\}$ of $\kappa' \cup \{\varsigma \supseteq \sigma\}$. We get:

$$\theta'' \overline{\mathcal{E}} = \theta'(\theta \overline{\mathcal{E}}) \qquad \tau_i \xrightarrow{\sigma'} \tau_f = \theta'(\theta\alpha \xrightarrow{\varsigma} \tau) \quad \text{and} \quad \sigma'' \supseteq \emptyset$$

Case of (app) The hypothesis is $\theta''\overline{\mathcal{E}} \vdash (\mathtt{e}_1\ \ \mathtt{e}_2) : \tau', \sigma'_1 \cup \sigma'_2 \cup \sigma'_3$. By the definition of the rule (app), this requires that,

$$\theta''\overline{\mathcal{E}} \vdash \mathtt{e}_1 : \tau'_2 \xrightarrow{\sigma'_3} \tau', \sigma'_1 \quad \text{and} \quad \theta''\overline{\mathcal{E}} \vdash \mathtt{e}_2 : \tau'_2, \sigma'_2$$

By induction hypothesis on $\mathtt{e}_1$, $(\theta_1, \tau_1, \sigma_1, \kappa_1) = \mathcal{I}(\mathcal{E}, \kappa, \mathtt{e}_1)$ and there exists θ'_1 satisfying κ_1 such that:

$$\theta''\overline{\mathcal{E}} = \theta'_1(\theta_1\overline{\mathcal{E}}) \qquad \tau'_1 = \tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_1\tau_1 \quad \text{and} \quad \sigma'_1 \supseteq \theta'_1\sigma_1$$

Since $\theta_1\overline{\mathcal{E}} = \overline{\theta_1\mathcal{E}}$ and $\theta_1\mathcal{E}$ is well-formed, by induction hypothesis on $\mathtt{e}_2$, we get

$$\mathcal{I}(\theta_1\mathcal{E}, \kappa_1, \mathtt{e}_2) = (\theta_2, \tau_2, \sigma_2, \kappa_2)$$

and there exists θ'_2 satisfying κ_2 such that:

$$\theta''\overline{\mathcal{E}} = \theta'_2(\theta_2(\theta_1\overline{\mathcal{E}})) \qquad \tau'_2 = \theta'_2\tau_2 \quad \text{and} \quad \sigma'_2 \supseteq \theta'_2\sigma_2$$

Let V be the set of free variables in $\overline{\kappa}_2(\theta_2(\theta_1\overline{\mathcal{E}}))$, $\overline{\kappa}_2\tau_2$ and $\overline{\kappa}_2\sigma_2$. Take α and ς new and define θ'_3 as follows:

$$\theta'_3 v = \begin{cases} \theta'_2 v, & v \in V \\ \tau', & v = \alpha \\ \sigma'_3, & v = \varsigma \\ \theta'_1 v, & \text{otherwise} \end{cases}$$

By this definition, θ'_3 satisfies κ_2 and we get:

$$\theta''\overline{\mathcal{E}} = \theta'_3(\theta_2(\theta_1\overline{\mathcal{E}})) \qquad \tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_3(\tau_2 \xrightarrow{\varsigma} \alpha) \quad \text{and} \quad \theta'_2\sigma_2 = \theta'_3\sigma_2$$

by definition of $\mathcal{I}$ every v in $\overline{\kappa}_1\tau_1$ is either fresh or in $fv(\overline{\kappa}_1\theta_1\overline{\mathcal{E}})$. Since $\theta'_3(\theta_2(\theta_1\overline{\mathcal{E}})) = \theta'_2(\theta_2(\theta_1\overline{\mathcal{E}})) = \theta'_1(\theta_1\overline{\mathcal{E}})$, for every v in $fv(\overline{\kappa}_1\theta_1\overline{\mathcal{E}})$, we have $\theta'_3(\theta_2 v) = \theta'_2(\theta_2 v) = \theta'_1 v$. For every fresh v, since $\theta_2 v = v$, we have $\theta'_3(\theta_2 v) = \theta'_3 v = \theta'_1 v$. Thus,

$$\tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_3(\theta_2\tau_1) \qquad \theta'_1\sigma_1 = \theta'_3(\theta_2\sigma_1)$$

Since θ'_3 satisfies κ_2 and $\theta'_3(\theta_2\tau_1) = \theta'_3(\tau_2 \xrightarrow{\varsigma} \alpha)$, by the lemma 3.17, there exists a substitution θ_3 such that $\theta_3 = \mathcal{U}_{\kappa_2}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha)$ verifying:

$$\theta_3(\theta_2\tau_1) = \theta_3(\tau_2 \xrightarrow{\varsigma} \alpha)$$

and such that $\theta_3\kappa_2$ is well-formed. By the definition of the algorithm, we get:

$$\mathcal{I}'(\mathcal{E}, \kappa, (\mathtt{e}_1\ \ \mathtt{e}_2)) = (\theta, \tau, \sigma_3, \kappa')$$

where $\theta = \theta_3 \circ \theta_2 \circ \theta_1$, $\tau = \theta_3\alpha$, $\sigma_3 = \theta_3(\theta_2\sigma_1 \cup \sigma_2 \cup \varsigma)$ and $\kappa' = \theta_3\kappa_2$. By the lemma 3.17, there exists a substitution θ' satisfying $\theta_3\kappa_2$ such that $\theta'_3 = \theta' \circ \theta_3$. We get:

$$\theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}}) \qquad \tau'_1 = \theta'(\theta_3(\tau_2 \xrightarrow{\varsigma} \alpha)) \quad \text{and} \quad \sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \theta'(\theta_3\sigma_3)$$

By definition of $\mathcal{I}$, we have that:

$$(\theta, \tau, \sigma, \kappa') = \mathcal{I}(\mathcal{E}, \kappa, (\mathtt{e}_1\ \ \mathtt{e}_2)) \quad \text{and} \quad \sigma = \mathit{Observe}(\overline{\kappa}'\theta\overline{\mathcal{E}}, \overline{\kappa}'\tau)(\overline{\kappa}'\sigma_3)$$

By definition, $\sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \theta'\sigma_3$ and by the lemma 3.3,

$$\sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \mathit{Observe}(\theta'(\theta\overline{\mathcal{E}}), \theta'\tau)(\sigma') \supseteq \mathit{Observe}(\theta'(\theta\overline{\mathcal{E}}), \theta'\tau)(\theta'\sigma_3)$$

Since $\theta'_3 \models \kappa_2$ and $\theta'_3 = \theta' \circ \theta_3$, then, by definition of κ', we get that $\theta' \models \kappa'$. By the lemma 2.10, $\theta' \circ \overline{\kappa}' = \theta'$. By the lemma 3.3, this implies:

$$\sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \theta'\,\mathit{Observe}(\overline{\kappa}'\theta\overline{\mathcal{E}}, \overline{\kappa}'\tau)(\overline{\kappa}'\sigma_3) = \theta'\sigma$$

We can conclude that $(\theta, \tau, \sigma, \kappa') = \mathcal{I}(\mathcal{E}, \kappa, (\mathtt{e}_1\ \ \mathtt{e}_2))$, $\theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}})$, $\tau' = \theta'\tau$ and $\sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \theta'\sigma$ □

3.7 Examples

By introducing some well-known examples of list processing procedures, we show that our type and effect system permits the assignment of the same type and effect to the functional and imperative implementations of some familiar list processing functions.

$$
\begin{array}{ll}
\texttt{nil} & : \forall\alpha.\mathit{list}(\alpha) \\
\texttt{cons} & : \forall\alpha\varsigma.\alpha \times \mathit{list}(\alpha) \xrightarrow{\varsigma} \mathit{list}(\alpha) \\
\texttt{car} & : \forall\alpha\varsigma.\mathit{list}(\alpha) \xrightarrow{\varsigma} \alpha \\
\texttt{null?} & : \forall\alpha\varsigma.\mathit{list}(\alpha) \xrightarrow{\varsigma} \mathit{bool} \\
\texttt{cdr} & : \forall\alpha\varsigma.\mathit{list}(\alpha) \xrightarrow{\varsigma} \mathit{list}(\alpha)
\end{array}
$$

List Processing Functions

We introduce the type $\mathit{list}(\tau)$ of immutable lists together with the following constant and functions for manipulating them: `nil` is the empty list. The predicate `null?` tests if a list is empty. The constructor `cons` pairs up an element of type α with a list of type $\mathit{list}(\alpha)$. The procedure `car` returns the first element of a list and `cdr` the rest.

Example Our first example is the function `fold`, that operates a function `f` over every elements of a list `l` and its intermediate result `i`.

```
(define fold (lambda (f i)
               (lambda (l) ((rec (loop l i)
                                 (if (null? l) i (loop (cdr l)
                                                       (f (car l) i))))
                            l i))))
```

By considering observable effects, implementing the function `fold` recursively or by a loop using temporary locatives does not affect its typing.

```
(define fold (lambda (f i)
               (lambda (l)
                 (let (result (new i))
                   (let (data (new l))
                     (until (null? (get data))
                       ((set result) (f (car (get data)) (get result)))
                       ((set data) (cdr (get data)))))
                   (get result)))))
```

Both implementations of the function `fold` have type $\forall\alpha\alpha'\varsigma\varsigma'\varsigma''.(\alpha \times \alpha' \xrightarrow{\varsigma} \alpha') \times \alpha' \xrightarrow{\varsigma'} (\mathit{list}(\alpha) \xrightarrow{\varsigma\cup\varsigma''} \alpha')$. Then, if one defines the function `reverse` for reversing the elements of a list by `(lambda (l) ((fold cons nil) l))`, the polymorphic type is $\forall\alpha\varsigma.\mathit{list}(\alpha) \xrightarrow{\varsigma} \mathit{list}(\alpha)$ with both implementations of `fold` ■

Example In the same vein, we consider the typing of two implementations of the function `map`. First we implement `map` by using `reverse` and imperative constructs.

```
(define map (lambda (f l)
              (let (r (new nil))
                (let (x (new l))
                  (until (null? (get x))
                    ((set r) (cons (f (car (get x))) (get r)))
                    ((set x) (cdr (get x)))))
                (reverse (get r)))))
```

The detail of the imperative implementation of the function `map` are similar to those of the function `fold` and we can also implement `map` by reusing the function `fold` defined above.

```
(define map (lambda (f l)
              ((fold (lambda (x r) (cons (f x) r)) nil) l)))
```

The same polymorphic type is assigned to both of them, by using our type and effect discipline. So, for instance, the application of `map` to the identity function and the empty list has the polymorphic type $\forall\alpha.list(\alpha)$ of `nil`.

```
(define nil1 (map (lambda (x) x) nil))
```

On the contrary, the application of `map` to `new` and `nil` has a monomorphic type $list(ref_\rho(\alpha))$, accounting for the use of the function `new` on a region ρ, with an observable effect $init(\rho, \alpha)$.

```
(define nil2 (map new nil))
```

3.8 Related Work

We have completely formalized our imperative typing discipline with effects and proved its correctness. Before concluding, we shall now address the related work and then to present a detailed comparison of our approach with it.

3.8.1 Weakening Type Variables

The first approach to control the typing of reference values has been introduced in the type system of [Tofte, 1987]. It is used in the current implementation of the language Standard ML [Milner & al., 1990]. This typing discipline consists of distinguishing the type of operations that create references. This is done, first, by defining a class of "weak" type variables which represent the type variables free in the type of initialized references and second by restricting the generalization of weak type variables on non-expansive expression.

Example In [Tofte, 1987], the body of a `let` construct is typed under the assumption that the operator `new` for allocating references has type scheme $\forall\alpha^*.\alpha^* \rightarrow ref(\alpha^*)$ where α^* is a weak type variable and cannot be substituted with a term that contains a non-weak type variable.

```
(let (ref-id (new (lambda (x) x)))
  ((get ref-id) 1))
```

The expression `(new (lambda (x) x))` above is expansive and thus, to be consistent with the use `((get ref-id) 1)` of `ref-id` in the body of the `let` construct, α^* must be instantiated to int ■

Expressing the type of a function that uses temporary references with weak type variables does not permit us to generalize any of the applications of that function. This makes a difference between the typing of procedures which depends on whether the style of their implementation is functional or imperative.

Example For instance, take the simple case the identity function `id` and what could be considered as its imperative variant: `rid`.

```
(define id (lambda (x) x))
(define rid (lambda (x) (get (new x))))
```

The application of our two variants of the identity function to `id` have different types.

```
(define id2 (id id))
(define id3 (rid id))
```

The type of `id2`, defined with the functional identity `id`, is generalizable. On the contrary, the type of `id3` cannot be generalized because it has weak type variables which come from the instance of `rid`'s type ■

Because the criterion for distinguishing expansive versus non-expansive expressions is simple, many expressions cannot be recognized as semantically non expansive. For example, the type of the function `(id new)` is weakly polymorphic, but a criterion of expansiveness does permit detection of generalizability.

3.8.2 Weakness Degrees for Type Variables

In order to detect precisely when references are created, Appel&Macqueen's implementation of Standard ML [Appel & Mac Queen, 1990] proposes an extension of Tofte's notion of weak type variables. In this system, type variables are associated to a weakness degree. The lower the degree is, the weaker the variable is. The formal description and soundness proof of this system can be found in [Hoang & al., 1992].

The weakness degree of a variable measures the number of function applications that must be performed before a reference whose type contains this variable is allocated. Variables that do not appear in the type of a `new` have an infinite degree. Variables that appear in the type of a `new` have a degree equal to the number of function abstractions that separate the introduction of the variable from the occurrence of the `new`. Variables with degree 0 are not generalizable. Variables with degree $n > 0$ are generalizable. Each function application decrements the degree of variables in the result type.

Example In Appel&MacQueen's type system, the function `k+new` below has type $\forall\alpha^2\alpha'^2.\alpha^2 \to \alpha'^2 \to ref(\alpha'^2)$. The partial application `k+new(1)` has type $\alpha'^1 \to ref(\alpha'^1)$, in which α' is still generalizable in Appel&Macqueen's system, since it has degree 1.

```
(define k+new (lambda (x) new))
(define new1 (k+new 1))
```

However, it is not generalizable in Tofte's system, because the expression `(k+new 1)` is expansive ■

The type system of Appel&Macqueen seems to be designed to answer the most evident problems encountered with the type system of Standard ML. This is, however, at the cost of a sightly more complicated notation, which forces the user to count the weakness degrees of the type variables he has to cope with and introduces additional dependencies between the syntax and the typing of programs.

The type system of Appel&Macqueen succeeds to syntactically circumvent the problem of expansiveness, but it does not take into account the scope locality of data, and thus, the examples of the previous section: `(id rid)` and `(rid id)`, cannot still be generalized.

As a conclusion, the type systems of Standard ML and its variants do not succeed in fully integrating imperative programming features to polymorphic functional languages, in that the imperative and functional implementation of the same function cannot be assigned equivalent and fully polymorphic types with them. There is, unfortunately, no simple and general solution for the programmer to avoid this problem, but to banish either functional or imperative programming styles.

In [Damas, 1985], the author proposes a sightly different approach to the problem of typing references, the soundness of which is disputed in [Tofte, 1987]. It consisted of annotating the type schemes of functions by the set of type variables possibly free in the types of the references allocated by the function. Unlike for the other approaches based on effect systems and on closure typing, these sets of type variables are not propagated using function types.

3.8.3 Typing Closures

In contrast with the previous work, the approach of closure typing, introduced in [Leroy & Weis, 1991], associates with the typing of a function syntactic information, which consists of the type of every identifier free in that function. This approach turns out to be quite effective in the issue of detecting the type of accessible reference values.

The type system of [Leroy & Weis, 1991] mainly differs from Milner's polymorphic type discipline $E \vdash \mathtt{e} : \tau$ by a different judgement, of the form $E \vdash \mathtt{e} : \tau/C$. Just as in effect systems [Lucassen & Gifford, 1988],

the type of functions, of the form $\tau \xrightarrow{L} \tau'$, does not only consists of the type of the argument τ and of the result τ', but also of a static information, L. This L is a label which is related to a sequence of types $\vec{\tau}$, called dangerous types, via a set C of constraints of the form $L \rhd \vec{\tau}$.

When a type τ/C is generalized, by $Gen(E/C)(\tau/C)$, the constraints associated with it in C are scanned so as to detect the type of references captured in τ. This generalization criterion was supported in [Leroy & Weis, 1991], but it is not conservative over ML, in that some expressions, which can be typed in ML, cannot be typed with this approach. This problem is reportedly illustrated by the example `capt-id` presented below, inspired from [Leroy & Weis, 1991].

Example As can be sketched by the following example, quoted from [Leroy & Weis, 1991], this change makes the closure typing approach conservative over ML.

```
(define capt-id (lambda (f)
                  (let (id (lambda (y)
                             (if true (lambda (z) y z)
                                      f)
                             y))
                    (id id))))
```

Let $\alpha \xrightarrow{L} \alpha/\emptyset$ be the type of `f` and α' be the type of `y`. The closure `(lambda (z) y z)` has a free variable: `y`. Hence, its type is $\alpha \xrightarrow{L} \alpha/\{L \rhd \alpha'\}$. The `if` construct forces the type of the occurence of `f` in `id` to match with that type. Because the type variable α' occurs in the type of `f`: $\alpha \xrightarrow{L} \alpha/\{L \rhd \alpha'\}$, the function `id`, which has type $\alpha' \xrightarrow{L'} \alpha'$ and constraint $\{L' \rhd \alpha \xrightarrow{L} \alpha, L \rhd \alpha'\}$, cannot be generalized over it ■

The improvements of the closure typing system, described in [Leroy, 1992], allow the generalization of `id` over α', as in ML, by making the generalization criterion $Gen(E/C)(\tau/C)$ unsensitive to free dangerous variables in the type environment E/C. These improvements allows the closure typing system to be conservative over ML.

However, the problem of capture, previously reported in [Leroy & Weis, 1991], does not completely vanish with the modification introduced in [Leroy, 1992] and there are still some examples in which this phenomenon appears. One is the function `id5` in the section 3.9.

The approach based on closure typing improves over the type discipline of Standard ML. However, the additional information needed for making it conservative over ML makes it depart a lot from the other. The representation of this information in the static semantics, via indirections between types and constraints, seriously impedes the intuitive understanding of the type system.

3.8.4 Typing Effects

In [Jouvelot & Gifford, 1991], the authors introduce the notion of algebraic reconstruction of type and effect in the presence of explicit polymorphic types. The system relates `plambda` expressions, which give explicit polymorphic types to the value of expressions, with algebraic constraints on effects: `Plambda` expressions must have `pure` effects.

Example By doing this, the system sometimes gives better results than other systems based on weak variables. For instance, the application `(id new1)` can be correctly recognized as pure by their system, and therefore its type can be explicitly generalized by

```
(plambda ((t type)(e effect))
  (the (subr (init) ((x (refof t))) (refof t))
    (id new1)))
```

The function `id` has no latent effect, and its argument is a pure expression. Annotating function types by their latent effect allows the system to detect when references are created more precisely ■

In this system, the generic functions that allocate temporary references are handled just as in Standard ML. Even though purely local, allocations are always witnessed in the latent effect. Hence, the type of these functions differs from the type of equivalent purely functional functions.

Example For instance, the following is accepted in Standard ML, but rejected in this effect system.

```
(let (id (begin (new nil)
                (lambda (x) x)))
  (id id))
```

The expression `(begin (new nil) (lambda (x) x))` has an initialization effect. Thus, its type cannot be generalized ■

In [Wright, 1992], the author presents a related type system which collects so called "typing effects". Typing effects consist of the set of type variables occuring in the type of allocated references. The table of section 3.9 shows that the properties of this type system are similar to the system presented in [Jouvelot & Gifford, 1991].

In contrast to a system based on algebraic reconstruction, which infers, in the exapmle above, that the effect of the expression bound to `id` is not pure, the system of [Wright, 1992] can detect that there is no typing effect on the type of `x`, so that the type of `id` is generalizable.

3.9 Comparison with the Related Work

In order to argue in favor of our system, the section 3.7 aims at demonstrating that the problem of integrating imperative features to a functional language is best viewed in terms of a type and effect system, by describing practical situations. The basic design goal of integration between functional and imperative paradigms is a fundamental aspect of our system. It is particularly well suited within a programming environment that supports separate compilation features and modular programming paradigms.

To give a detailed comparison of our system with the related work, the criterion of expressiveness seems at very first sight best suited, because it is a formal criterion. The relative expressiveness of a type system with respect to another defines itself as the capability of accepting strictly more programs. Such a proposition requires a formal proof and in most cases, it is thus easier to show the contrary by giving a counter example. In practice, it turns out that the criterion of expressiveness is not appropriate and many examples show that there is usually no proper inclusions between type systems.

3.9.1 Comparative Examples

In this section, we will now present a series of more or less sophisticated examples, adapted from a survey paper on this subject [O'Toole, 1990], and from [Leroy, 1992], that establishes the known frontiers between the related type systems.

```
(define id1 (let ((x (id 1)))
              rid))

(define id2 (lambda (y)
              ((rid id) y)))

(define id3 ((nop rid) id))
```

In these three examples, we write `rid` the "imperative" version of the identity function, defined in the section 3.8.1, and `id` for the usual definition of the identity functional. We also use the function `nop` defined as below.

```
(define nop (lambda (f)
              (lambda (x)
                (let ((g (lambda (y) (f x))))
                x))))
```

The next example, quoted from [Leroy, 1990], shows that the observation of effects can be useful for removing type dependencies introduced by otherwise dead-code. Our inference system generalizes the type of `id1` below, contrarily to the ones defined in [Leroy, 1990] or [Wright, 1992], which are not able to deal with the spurious allocation effect.

```
(lambda (z)
 (let ((id4 (lambda (x)
              (if true (lambda (y) (begin (new x) y))
                       z)
              x)))
   (id4 id4)))
```

Nonetheless, one can force such an effect to be observable, as is shown in this other very sophisticated example. An occurrence of the type of `y` appears on the arrow of the function type for `g` and is constrained to match the type of `f` which occurs in the context of `id5`. As a consequence, the type of `id5` cannot be generalized.

```
(lambda (f)
  (let ((id5 (lambda (y)
               (let ((r (new y)))
                 (if true (lambda (z) (if true r (new y))
                                      z)
                          f))
               y)))
    (id5 id5)))
```

In addition to this comparison, the example of recursive typing presented in section 3.5 (the function `eta-ref` below requires such a recursive typing) is well typed using the system of [Leroy, 1992]. Note that it would have been in ours, if we had chosen to make indirections between types and effects explicit in the static semantics, using constraint sets, as is done in the inference algorithm.

```
(define eta-ref (lambda (f)
                  (let (r (new f))
                    (if true (lambda (x) (get (if true r (new f)))
                                         x)
                             f))))
```

3.9.2 Benchmarks

The following benchmark summarizes the discussion above and suggests that our type and effect discipline favorably competes with some earlier polymorphic type generalization policies.

Example	[Tofte]	[Appel]	[Jouvelot]	[Wright]	[Leroy, 1990]	[Leroy, 1992]	section 3.1
`id1`	no	yes	yes	yes	yes	yes	yes
`id2`	yes	no	yes	yes	yes	yes	yes
`id3`	no	no	yes	yes	yes	yes	yes
`id4`	yes	yes	no	no	no	yes	yes
`id5`	yes	yes	no	no	no	no	no
`(id1 id1)`	no	no	no	no	yes	yes	yes
`(id2 id2)`	no	no	no	no	yes	yes	yes
`(id3 id3)`	no	no	yes	yes	yes	yes	yes
`(id4 id4)`	no	no	no	no	no	yes	yes
`(id5 id5)`	no	no	no	no	no	no	no

Leroy's system, based on closure typing, essentially differs from ours, based on effect inference, in that it associates functions with a static information: the type of their free variables. Type generalization then relies on the process giving chase to dangerous types. This resembles the process of garbage collection, which gives chase to referenced values and marks them before collecting the other.

In our approach, type generalization is expressed in a much more natural way, because it states what information is important to it: initialized reference values. Instead of chasing for every possibly referenced values at any time, as in a closure-typing system, we define a notion of accessibility in the static semantics, which is represented by an observation criterion.

This explains the differences reported in the examples `eta-ref`, above, and `fake-ref`, below, adapted from [Leroy, 1992] (Another example is the function `f`, defined section 3.3.1).

```
(define ref-id (lambda (x) (get x) x))
(define fake-ref (ref-id ((rec loop (x) (loop x)) 0)))
```

A type system based on closure typing cannot detect the fake character of the reference introduced in this example. A type system based on effect inference can. It detects that no initialization effect occur.

3.10 Future Work

An appealing direction for further extensions would be the treatment of first-class continuations. Continuations are objects that allow programs to capture current state of their evaluation, using the higher-order `callcc` operation: "call with current continuation", and to manipulate it, using the `throw` construct. These two very simple and general operations define sophisticated control structures that may be needed to implement interleaving or backtracking mechanisms, for instance.

Continuation objects were originally introduced in the language Scheme [Rees & al., 1988] and then, but not without trouble, in the implementation of Standard ML [Duba & al., 1991]. Continuation values where first proposed by [Duba & al., 1991] as an extension of Standard ML implemented by an abstract data type. Latter, the implementation of continuations in Standard ML of New Jersey [Appel & Mac Queen, 1990] was shown unsound in the presence of type polymorphism [Harper & Lillibridge, 1992].

An accessible continuation value allows restarting the evaluation of an expression in a context which is different from the context in which it was typed. Thus, it appears very natural to restrict polymorphism for continuations in the same vein as for references: accessible continuations shall remain monomorphic.

In our typing discipline with effects, the typing of continuations can be integrated in a manner very similar to reference values by the following static semantics.

$$
\begin{aligned}
\mathit{TypeOf}[\![\texttt{callcc}]\!] &= \forall\alpha\rho\varsigma\varsigma'.(\mathit{cont}_\rho(\alpha) \xrightarrow{\varsigma} \alpha) \xrightarrow{\varsigma'\cup\varsigma\cup\mathit{comefrom}(\rho,\alpha)} \alpha \\
\mathit{TypeOf}[\![\texttt{throw}]\!] &= \forall\alpha\alpha'\rho\varsigma\varsigma'.(\mathit{cont}_\rho(\alpha) \xrightarrow{\varsigma} \alpha) \xrightarrow{\varsigma'\cup\varsigma\cup\mathit{goto}(\rho,\alpha)} \alpha'
\end{aligned}
$$

Static Semantics for Continuations

Above, we define $\mathit{cont}_\rho(\tau)$ to be the type for continuation values and the effects $\mathit{comefrom}(\rho,\tau)$, for capturing the current continuation, and $\mathit{goto}(\rho,\tau)$, for invoking a continuation, as in [Jouvelot & Gifford, 1989].

The addition of continuations to our language does not, however, show up as a very natural extension as far as the dynamic semantics and the proofs of consistency are concerned. The dynamic semantics needs to be completely reformulated in terms of a continuation based semantics, as in [Duba & al., 1991], or in terms of a reduction semantics, as in [Wright & Felleisen, 1992].

In [Leroy, 1992], following along the lines of [Duba & al., 1991], the author gives a "weak soundness" result for the extension of its typing discipline to continuation objects, which states that "well typed expressions cannot evaluate to *wrong*". The reduction semantics [Felleisen & Friedman, 1989] still permits the formulation of a strong soundness result for continuations in the typing discipline of Standard ML [Wright & Felleisen, 1992].

3.11 Conclusion

Type inference [Milner, 1978] is the process that automatically computes the type of expressions in programming languages. It allows programmers to let types implicit in programs. The burden of writing types in a program does not, however, completely disappear with type inference.

They are, indeed, a number of situations where the programmer is required to have a good understanding of the type system. The most common situation is when a type error is reported by the compiler. The programmer must then understand the message, make a diagnostic and locate the error in the program. Developing large programs is made easier by decomposing them into modules that export a few identifiers to the remainder of the program following a strict protocol. Thus, programmers must also be capable of writing type expressions, when declaring data types or when writing module interfaces.

At very first sight, writing types raise some difficulties related to the expressive power of the type system. The more informative the type system is, the more difficult writing types is. What helps to motivate the programmer is how effective it is to give such information. Effect systems provide this additional motivation by giving support for program optimizations.

Imperative features are required to make functional programming realistic, but integrating polymorphic typing in an imperative language appears problematic. Solving this tension is best viewed by using an effect systems. Effect systems answer the lack of specification for polymorphic typing in the presence of effects by approximating state transformations.

Using effects, our typing discipline reconstructs the principal type and the minimal observable effects of expressions. We use effect information to control type generalization. We use an observation criterion to precisely delimit the scope of side-effecting operations. Our observation criterion generalizes the abstraction properties of functions in the presence of a state. A function can use a mutable object locally in a given region without making mention of this region outside. Altogether, this allows type generalization to be performed in `let` expressions in a more efficient and uniform way than previous systems.

The initial design goal of polymorphic effect systems [Lucassen & Gifford, 1988] was to safely integrate functional and imperative constructs. We showed how effect systems can also be put to work for solving the problem of polymorphic type reconstruction in the presence of imperative constructs. Our typing discipline permits full integration of an imperative programming style in a polymorphic functional language.

Chapter 4

Compiling FX on the CM-2

4.1 Introduction

The functional and imperative programming paradigms are often integrated together within sequential languages such as Common Lisp [Steele, 1990], Scheme [Rees & al., 1988] and Standard ML [Milner & al., 1990]. In such languages, implementors must exert care when designing code optimizers since side effects inhibit most of the nice properties of pure functional languages which are put at work in code transformations.

Going from sequentiality to parallelism, issues get significantly more complicated, both at the programmer and implementor levels. Concomitant use of side effects and parallelism leads to nondeterminism, which makes program understanding and debugging difficult because of the non-reproductibility of results. Restricting parallel programs to be deterministic, as advocated in [Steele, 1990], is a way of making parallel program design in higher-order imperative languages a more manageable task. Based on the concept of an effect system [Lucassen & Gifford, 1988, Lucassen, 1987], we present here a compile-time technique that enforces such deterministic constraints and prove its effectiveness by describing a prototype compiler that targets the FX programming language [Gifford & al., 1987] to the Connection Machine[1] (referred to as CM-2).

The purpose of the FX/CM compiler is to demonstrate the effectiveness of program analysis and code transformations based on type and effect information for high-level higher-order imperative languages. Our compiler uses the type and effect system of chapter 2 to determine when operations on vectors are amenable to data parallelism in the presence of both side effects and higher-order functions. The absence of side-effects for an operation mapped on every element of a vector guarantees that its execution in parallel will not cause interferences. Such operations are run in parallel while others are conservatively limited to sequential execution on the CM-2 front end. Our compiler uses regions to discover when the lifetime of locally allocated data structures is compatible with the memory model of the CM-2, which encourages the allocation of parallel vectors in the stack.

An implementation of these compile-time techniques has been integrated in the FX system, providing a CM-2 compiler that generates *Lisp[2] code. Test programs have been run on both a *Lisp simulator [*Lisp, 1987] and a CM-2 to evaluate the practicality and the performance of our approach.

Plan

After presenting in section 4.2 the vector module of the FX language, we give in section 4.3 a brief overview of the CM-2 architecture and its object language *Lisp. We survey in section 4.12 the related work and present in section 4.4 the essential design ideas of our analysis and code generation techniques. We discuss in section 4.11 the interesting implementation issues before concluding in section 4.13. This work was partially supported by the MIT contract GC-R-117153.

[1] Connection Machine is a registered trademark of Thinking Machines Corporation.
[2] *Lisp is also a trademark of Thinking Machines Corporation.

4.2 Operations on Vectors

Since we are interested in studying the practical applications of effect systems in order to implement data parallelism, we focus on the FX module describing operations on vectors so as to allow data parallel algorithms to be easily expressed. Vectors are a specific data structure represented by the abstract data type $vector_\rho(\tau)$ which denotes mutable arrays allocated in the region ρ whose elements are of type τ. In the following, we do only give the lower bound of the latent effect of functions.

op ::=			
	`make-vector`	initialization	$\tau \times int \xrightarrow{init(\rho)} vector_\rho(\tau)$
	`vector-ref`	dereference	$vector_\rho(\tau) \times int \xrightarrow{read(\rho)} \tau$
	`vector-set!`	assignment	$vector_\rho(\tau) \times int \times \tau \xrightarrow{write(\rho)} unit$
	`vector-length`	length	$vector_\rho(\tau) \xrightarrow{read(\rho)} int$

Basic Vector Operations

Basic vector operations first comprise the initialization, `(make-vector v n)`, which allocates a vector of length `n` initialized to the value `v`. The operation `(vector-ref v n)` dereferences the `n`th element of the vector `v`. The assignment of the `n`th element of the vector `v` to the value `e` is performed by the operation `(vector-set! v n e)`. The operation `(vector-length v)` gets the length of a vector `v`.

op ::=			
	`identity`	identity permutation	$int \xrightarrow{init(\rho)} vector_\rho(int)$
	`permute`	regular permutation	$vector_\rho(int) \times vector_{\rho'}(\tau) \xrightarrow{read(\rho) \cup read(\rho') \cup init(\rho'')} vector_{\rho''}(\tau)$
	`cshift`	circular shift	$int \times vector_\rho(\tau) \xrightarrow{read(\rho) \cup init(\rho')} vector_{\rho'}(\tau)$
	`compress`	compression	$vector_\rho(bool) \times vector_{\rho'}(\tau) \xrightarrow{read(\rho) \cup read(\rho') \cup init(\rho'')} vector_{\rho''}(\tau)$

Permutation Operations

Permutations permit the rearrangement of vectors. When vectors are implemented by distributed data structures, like on the CM-2, permutations implement inter-process communications. The operation `(identity n)` returns the identity permutation from `1` to `n`. The operation `(permute p v)` performs the rearrangement of `v` according to the permutation `p`. The operations `(cshift n v)` and `(eoshift n v v')` are the usual circular and end-off shift permutations. The operations `(compress s v)` and `(expand s v v')` are as follows:

```
fx> (compress #(true false true) #(1 2 3))
= #(1 3)
fx> (expand #(true false true false true) #(1 2 3) #(0 0 0 0 0))
= #(1 0 2 0 3)
```

In the operation `compress`, `s` and `v` should be the same size. `Compress` selects and concatenates the elements of v_i such that the s_i are true. In the operation `(expand s v v')`, `s` is the same length as `v'` and bigger than `v`. When s_j is true for the i^{th} time, v_i is selected, v'_j otherwise.

op ::=			
	`vector-map`	mapping	$(\tau \xrightarrow{\sigma} \tau') \times vector_\rho(\tau) \xrightarrow{\sigma \cup read(\rho) \cup init(\rho')} vector_{\rho'}(\tau')$
	`vector-scan`	scanning	$(\tau \times \tau \xrightarrow{\sigma} \tau) \times vector_\rho(\tau) \xrightarrow{\sigma \cup read(\rho) \cup init(\rho')} vector_{\rho'}(\tau)$
	`vector-reduce`	reduction	$(\tau \times \tau \xrightarrow{\sigma} \tau) \times vector_\rho(\tau) \xrightarrow{\sigma \cup read(\rho)} \tau$

Mapping Operations

In addition to the standard Scheme-like basic vector operations, the current vector module supports the mapping and reduction of first-class functions like [Blelloch, 1990] and the Fortran90 array extensions [Fortran90].

```
fx> (vector-map not #(true false true false true))
= #(false true false true false)
```

The expression `(vector-map f v)` applies the unary higher-order function `f` to every element of the vector `v` and returns the vector with the successive applications of `f`.

```
fx> (vector-scan and #(true false true false true))
= #(true false false false false)
fx> (vector-reduce and #(true false true false true))
= false
```

The expression `(vector-scan f v)` and `(segmented-scan f s v)` sum up the binary higher-order function `f` over every element of the vector `v` and returns the vector of the successive applications of `f`. In `segmented-scan`, summation is reset at v_i if s_i is false. In `vector-reduce`, the sum is returned.

4.3 *Lisp and the Connection Machine

The architecture of the CM-2 is based on the SIMD model (Single Instruction Multiple Data). It is composed of up to 64k processing elements, wrapped by groups of 32 processors and local memory units, connected into a global hypercube communication network. A front-end workstation issues instructions and transfers data in a time-step fashion to the CM-2.

*Lisp is an extension of Common Lisp [Steele, 1990] that implements the PARallel Instruction Set (PARIS) and supports the data structure *pvars* (parallel variable). A pvar is a vector whose elements are allocated in the memory of every processing unit of the CM-2. In contrast to the usual implementation of vectors in Common Lisp systems, pvar components are unboxed values of fixed size such as booleans, fixnums or floats. The type of every pvar manipulated in the *Lisp program must be declared by the programmer.

The local memory of each processing element is divided into an heap area and a stack area. By default, a *Lisp pvar expression is allocated in the stack frame of the current call.

Example The following *Lisp expression distributes the factorial of 5 on every active processing element in an unboxed pvar coerced to the size of a `fixnum`.

```
*lisp> (coerce!! (fact 5) '(pvar fixnum))
= #<pvar x :general *default-vp-set* (1024)>
```

It is expanded, by the *Lisp compiler, into the the following piece of Common Lisp code, which performs calls to the appropriate PARIS instructions.

```
(let* ((slc::stack-field (cm:allocate-stack-field 31))
       (#:arg-1 (fact 5)))
  (declare (type slc::cm-address slc::stack-field))
  (slc::move-signed-constant slc::stack-field #:arg-1 31)
  (slc::allocate-temp-pvar :type :signed
                           :length 31
                           :base slc::stack-field
                           :constant-value #:arg-1))
```

The preceding sequence of PARIS instructions allocates a temporary pvar of 31 bits at the stack address `slc::stack-field` in the current stack frame. In this example, a stack-field, `slc::stack-field`, with the size of an integer is first allocated with the operation `(cm:allocate-stack-field 31)`. Then, the value of `(+ x y)` is computed and bound to `#:arg-1`. The variable is statically declared `slc::stack-field` to be of type `cm-address`. The `31` bytes of the value of `#:arg-1` are placed on the CM-2 at the address `slc::stack-field` by `slc::move-signed-constant`. Finally, the address `slc::stack-field` is boxed with a temporary pvar header allocated with the operation `slc::allocate-temp-pvar` ■

Overview of *Lisp

This section introduces the most important features and operations of *Lisp used in our compiler. The reader may find in [*Lisp, 1987] the complete definition of the language *Lisp.

In *Lisp, vectors are represented by pvars. There are several ways of allocating pvars. The most common way is to use temporary allocation, using the operation `!!` or `coerce!!`, described in the previous example. But it is the most unsafe. In *Lisp, pvar can also be explicitly stack allocated by using the construct `*let`.

```
*lisp> (*let ((x (coerce!! (+ 2 2) '(pvar fixnum))))
         (declare (type (pvar fixnum) x))
         (*deallocate (allocate!! x)))
= nil
```

This expression allocates the space for a `fixnum` on the stack of every PEs of the CM-2 and then moves the value `4` to this address on every PE using the operation `coerce!!`. Note that declaring the type of `x` is inevitable for the expression to be correctly compiled. The storage management of pvar expressions in the heap is also explicit. The operations that performs heap allocation and reclamation are `allocate!!` and `*deallocate` respectively.

```
*lisp> (+!! (!! 1) (!! 1))
= #<pvar x :general *default-vp-set* (1024)>
```

For arithmetic computations, the *Lisp system extends Common Lisp's generic operations on numbers. They are implemented by the so-called *bang-bang* functions. For example, the operation `+!!` is the equivalent to `+`, etc. *Bang-bang* functions operate in parallel on every component of their two pvar arguments and return a pvar.

4.4 Overview of the Compiler

In this chapter, we introduce a series of new compile time techniques which, based on our type and effect inference system, determine when the use of operations on vectors is actually amenable to data-parallel execution (no inhibiting side-effects) on the CM-2 (no unimplementable parallel operations).

In FX programs are implicitly typed, may have side-effects and may use first-class functions. Our compiler generates *Lisp programs that use pvars with explicit typing, explicit parallelism, explicit management of stack and heap storage and explicit name space assignment. *Lisp uses multiple name spaces for function and value identifiers.

In the static semantics of FX, every expression is associated with its type and effect. The criterion of parallelizability of expressions is based on type and effect. Parallelizable expressions must manipulate scalar data structures and have no side-effects.

Parallelizable FX functions are translated into specific data structures, noted f-structure, built with the operator `make-f-struct`, which consists of a tagged pair of functions, the sequential version (the `f-seq` component) and the parallel one (the `f-par` component), operating upon pvars.

The compiler translates FX vector operations by invoking appropriate runtime library macros to implement the operations on unboxed pvars for the corresponding vectors operations.

Even though FX is a strongly typed language, the addition of `let`-bound identifiers introduces generic polymorphism. The presence of type and effect variables requires the use of run-time type dispatch for vector operations.

We describe below the compiler memory allocation strategy, the compilation schemes for front-end and CM expressions (restricted to a simplified language), some optimization techniques and finally the management of let constructs and multiple namespaces.

4.5 Vector Allocation

Vectors are represented by pvars. Vectors with scalar components, such as boolean or integer, are implemented by unboxed pvars. Vectors with non-scalar components, such as lists or other vectors, are represented

by boxed pvars, called *front-end* pvars, which are pvars of addresses to objects that reside on the front-end i.e. front-end objects. Since *Lisp is dynamically typed, every pvar is associated on the front-end with a description header giving its actual address on every processing element and the size and type of its components.

Temporary allocation of pvars is preferred wherever possible in the generated *Lisp code in order to avoid the overhead of superfluous heap allocation. The FX compiler uses type and effect to decide when a returned value must be explicitly moved to the heap.

Definition 4.1 (Heap Allocation Criterion) *A vector operation* `(op` e_1 ... e_n`)` *that performs the allocation of a vector, of type* $vector_\rho(\tau)$*, must allocate its value in the heap in one of the following circumstances:*

- *The vector operation is the argument of an application expression* `(e [])` *whose type is* $ref_{\rho'}(\tau')$*,* $vector_{\rho'}(\tau')$*,* $list(\tau')$*, or* $\tau'' \xrightarrow{\sigma} \tau'$*. The region* ρ *of the vector argument occurs free in* τ' *or* σ*.*
- *The vector operation occurs in the body of a lambda expression of the form* `(lambda (x) C[])` *whose type is* $\tau'' \xrightarrow{\sigma} \tau'$*. The region* ρ *of the vector operation occurs free in* τ' *or in a subterm* τ''' *of* τ''*, be it a data structure* $ref_{\rho'}(\tau''')$*,* $vector_{\rho'}(\tau''')$ *or* $list(\tau''')$*.*

If the pvar outlives the stack frame it is allocated into, i.e. if it can be referenced in its lexical environment, it must be moved in the heap with an explicit call to `allocate!!`. The criterion presented above syntactically controls the cases in which this situations may happen and that can actually be checked by using the effect system.

Example In the following expression, the allocated vector escapes from the scope of its allocation according to the previous criterion. Thus, it has to be heap allocated.

```
(new (make-vector true 1024))
```

Typing this expression produces the following derivation. It appears that the criterion 4.1 applies to the region ρ in which the result of the vector operation is allocated.

$$\mathcal{E} \vdash \texttt{new} : vector_\rho(bool) \xrightarrow{init(\rho')} ref_{\rho'}(vector_\rho(bool)), \emptyset$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash \texttt{true} : bool, \emptyset \\ \mathcal{E} \vdash \texttt{1024} : int, \emptyset \\ \mathcal{E} \vdash \texttt{make-vector} : bool \times int \xrightarrow{init(\rho)} vector_\rho(bool), \emptyset\end{array}}{\mathcal{E} \vdash \texttt{(make-vector true 1024)} : vector_\rho(bool), init(\rho)}$$

$$\frac{}{\mathcal{E} \vdash \texttt{(new (make-vector true 1024))} : ref_{\rho'}(vector_\rho(bool)), init(\rho) \cup init(\rho')}$$

In *Lisp, such an operation is compiled by the following code, which consists of calls to macros in the FX runtime library.

```
(fx::new (fx::bool-heap-allocate (fx::bool-make-vector t 1024)))
```

The function `fx::new` returns a reference to its argument. The macro `fx::bool-heap-allocate` performs heap allocation of a boolean pvar. The macro `fx::bool-make-vector` allocates and returns a temporary boolean pvar as part of the current virtual processor set (VP set). The PEs from 1 to 1024 are activated and the value `t` is distributed to them. The value of the pvar on the other PEs is unspecified ■

4.6 Runtime Library

The runtime library provides the *Lisp functions and macros which implement the FX vector module. Every vector operation in FX corresponds to specialized *Lisp macros and a generic function in the runtime library, defined as follows.

- *Lisp macros implement the data-parallel polymorphic FX operations for every type of unboxed pvars: booleans, characters, integers, reals and complex.

Example For example, the operation `vector-ref` on an integer vector expression `v` and an index `i` and is translated to `(int-vector-ref v i)`, which then expands to:

```
(the fixnum (pref (the (pvar fixnum) v) (the fixnum i)))
```

The operation is also specificly implemented on front-end pvars by the appropriate macro (it is the macro `front-end-vector-ref` in this example) ■

- The generic function implements the operation for every type of pvars (in the example, `vector-ref`). It uses a dispatch construct depending on pvar headers to check the actual type of pvar arguments at run time and to call the appropriate specialized macro.

By default, all global vector operations are performed in parallel. The FX operations that accept higher order functions, such as `vector-map` or `make-permutation`, obey a different rule which is that there front-end version, e.g. `front-end-vector-map` or `front-end-make-permutation`, implement the sequential version on generic pvar arguments.

Finally, note that *Lisp operations on pvar are restricted to the set of active PEs or VP set. Similarly, parallel vector operations are limited in FX to their actual size. In the runtime library, this is implemented by the macro `(with-context-of e e′)`, which disables the processing elements on which the pvar expression `e` is not defined during the execution of `e′`.

4.7 Sequential Code Generation

Having briefly presented *Lisp, our FX vector module and the *Lisp runtime library corresponding to it, we can now describe the *Lisp code generation scheme for sequential expressions. The input of the compiler is an expression `te` which consists of an FX expression `e`, its principal type τ and its minimal effect σ.

`te` ::=	`x`:τ |	value identifier
	`(lambda (x`:τ`) te)`:τ |	abstraction
	`(op te)`:τ!σ |	operation
	`(te te)`:τ!σ |	application
	...	and so on

Syntax of Typed FX Expressions

The sequential code generation scheme is specified by an algorithm, *SC*, which relates a typed FX expression `te`, i.e. that was successfully typed checked by the algorithm of chapter 2, with the *Lisp code c that correspond to it in a given compilation context $\vec{\tau}$. This compilation context $\vec{\tau}$ serves to determine when vector allocations must be performed in a heap and consists of a sequence of types. $\vec{\mathtt{x}}$ is the sequence of parallelizable predefined functions, such as `+`, `-`, etc. We use the marks $[\![]\!]$ to delimit the syntactic objects passed to or returned from a function.

$$SC[\![\mathtt{te}]\!]\vec{\tau}\vec{\mathtt{x}} = c$$

Identifiers

The compiler translates a lambda-bound identifier `x` to `x`. An identifier that appears in $\vec{\mathtt{x}}$ is a predefined parallelizable function and is thus compiled by an f-structure.

$$SC[\![\mathtt{x}:\tau]\!]\vec{\tau}\vec{\mathtt{x}} = \text{if } \mathtt{x} \in \vec{\mathtt{x}} \text{ then } [\![\mathtt{(make\text{-}f\text{-}struct\ x\ x!!)}]\!] \text{ else } [\![\mathtt{x}]\!]$$

Compilation of Identifiers

Abstraction

To translate an abstraction, the compiler first compiles the body of the source FX abstraction. Then, if the lambda-abstraction can be parallelized according to the predicate PF defined below, an f-structure initialization is generated which pairs up the sequential version of the function, generated by SC, with its parallel version, generated by PC (presented in the next section).

$$
\begin{array}{l}
SC\,[\![\texttt{(lambda (x:}\tau\texttt{) te):}\tau \xrightarrow{\sigma} \tau'[\![\;\vec{\tau}\,\vec{\texttt{x}} = \\
\quad \text{let } d_1 = STD[\![\texttt{x:}\tau]\!] \\
\qquad c_1' = SC[\![\texttt{te}]\!](\vec{\tau}.\tau.\tau')\vec{\texttt{x}} \\
\qquad c_1 = [\![\texttt{\#'(lambda (x) } d_1 \;\; c_1'\texttt{)}]\!] \\
\quad \text{in if } PF(\tau \xrightarrow{\sigma} \tau') \\
\qquad \text{then let } (\texttt{f}_i\texttt{:}\tau_i)_{i=1,\ldots n} = \{\texttt{f}_i\texttt{:}\tau_i \in fi[\![\texttt{te}]\!] \setminus \texttt{x:}\tau \mid \tau_i = \tau_i' \xrightarrow{\varsigma_i \cup \sigma_i} \tau_i'' \wedge \varsigma_i \in \sigma\} \\
\qquad\quad d_2 = PTD[\![\texttt{x:}\tau]\!] \\
\qquad\quad c_2' = PC[\![\texttt{te}]\!](\vec{\texttt{x}}.\texttt{x:}\tau) \\
\qquad\quad c_2 = [\![\texttt{\#'(lambda (x!!) } d_2 \;\; c_2'\texttt{)}]\!] \\
\qquad\quad \text{in } [\![\texttt{(if (and (f-struct-p f}_1\texttt{)}\ldots\texttt{(f-struct-p f}_n\texttt{))} \\
\qquad\qquad\quad \texttt{(make-f-struct } c_1 \;\; c_2\texttt{)} \\
\qquad\qquad\quad c_1\texttt{))}]\!] \\
\qquad \text{else } c_1
\end{array}
$$

Compilation of Abstraction

If the function is not parallelizable then the compiler returns the sequential version c of the function. The algorithms $STD[\![]\!]$ and $PTD[\![]\!]$ respectively generate the sequential and parallel type declarations. The compiler uses the following static criterion, PF, based on the type of the function, to determine if it can be parallelized.

Definition 4.2 (Parallelizability Criterion) *A lambda-expression of type $\tau \xrightarrow{\sigma} \tau'$ is parallelizable (satisfies the criterion PF) if and only if its latent effect σ is $\emptyset$ or a union of effect variables, and if the types τ and τ' are scalar data types, type variables or function types that satisfy the criteria $PF(\tau \xrightarrow{\sigma} \tau') = PA(\tau) \wedge PA(\tau') \wedge PE(\sigma)$, where*

$$
\begin{array}{lll}
PA(\tau) = \textit{case } \tau \textit{ of} & & \\
\quad bool \mid int \mid real \mid \alpha & \Rightarrow true & \\
\quad \tau' \xrightarrow{\sigma} \tau'' & \Rightarrow PF(\tau) & \\
\quad otherwise & \Rightarrow false &
\end{array}
\qquad
\begin{array}{lll}
PE(\sigma) = \textit{case } \sigma \textit{ of} & & \\
\quad \emptyset \mid \varsigma & \Rightarrow true & \\
\quad \sigma \cup \sigma' & \Rightarrow PE(\sigma) \wedge PE(\sigma') & \\
\quad otherwise & \Rightarrow false &
\end{array}
$$

This criterion has both compile-time and runtime aspects. Its compile-time aspect can be informally justified in the following way. First, the lambda expression must have no side effects: no write effects $write(\rho)$ must occur in σ, since they could generate non-determinism at run time. Also, no initialization $init(\rho)$ or read effects $read(\rho)$ may appear, as they would indicate that the function allocates or manipulates non-scalar values which are unimplementable on the Connection Machine. Second, the types τ and τ' must be scalar variables or function types.

Runtime checks are only required in the presence of effect variables in σ, to distinguish whether these effect variables actually are $\emptyset$ or not in each given instance. These effect variables are introduced by the latent effects of higher-order functions, other lambda-bound function identifiers. However, we known that every pure first-order function is compiled as an f-structure. Thus, we use the predicate `f-struct-p` to decide at runtime whether the free function identifiers of a lambda-abstraction actually correspond to other f-structures. When this condition is met, an f-structure is returned.

Example For instance, the code generated for `(lambda (g) (lambda (x) (g x)))` is as below (The identifier `id` is fresh and the *Lisp function `*funcall` applies a function value operating on pvars to its arguments and is used inside the parallel code.)

```
(lambda (g)
  (let ((id #'(lambda (x)
                (funcall (if (f-struct-p g) (f-seq g) g) x))))
    (if (not (f-struct-p g)) id
        (make-f-struct id #'(lambda (x!!) (*funcall (f-par g) x!!))))))
```

■

Application

As already illustrated by the previous example, the general compilation scheme for an application is, first, to translate both subexpressions and, then, to generate the *Lisp code that checks at runtime whether the called function is a f-structure.

$SC\,[\![\texttt{(te te')}:\tau''!\sigma'']\!]\vec{\tau}\vec{\texttt{x}} =$
  let `f` fresh
    $c = SC[\![\texttt{te}]\!]\vec{\tau}\vec{\texttt{x}}$
    $c' = SC[\![\texttt{te}']\!]\vec{\tau}\vec{\texttt{x}}$
  in $[\![$`(let ((f` c`)) (funcall (if (f-struct-p f) (f-seq f) f)` c'`))`$]\!]$

Compilation of Application

Simplifications

More efficient compilation mechanisms for vector operations are given in the subsection 4.9. However, simple syntactic rewriting rules can be used to already improve the code generated by the previous technique. They are:

`(f-seq (make-f-struct` c c'`))` $\Rightarrow$ `c` `(funcall #'`c c'`)` $\Rightarrow$ `(`c c'`)`
`(f-par (make-f-struct` c c'`))` $\Rightarrow$ c' `(*funcall #'`c c'`)` $\Rightarrow$ `(`c c'`)`

For instance, the combination of boxing/unboxing of functions, done via `make-f-struct`, `f-seq` and `f-par`, can most of the time be simplified. This is also the case with the combination of *Lisp special forms `#'` and `[*]funcall`.

`(f-struct-p (make-f-struct` c c'`))` $\Rightarrow$ `t` `(if t` c c'`)` $\Rightarrow$ c
`(f-struct-p #'(lambda (x)` c`))` $\Rightarrow$ `nil` `(if nil` c c'`)` $\Rightarrow$ c'

In the same manner, we can simplify many runtime tests that are performed using `f-struct-p` and reduce the related `if` expressions.

`(let ((id (make-f-struct` c c'`))) ` c''`)` $\Rightarrow$ c''`[(make-f-struct` c c'`)/id]`
`(let ((id #'(lambda (x)` c`))) ` c'`)` $\Rightarrow$ c''`[#'(lambda (x)` c`)/id]`

Let-bindings introduced by our tiny compiler can also be safely inlined. Then, handling the substituted expressions reduces itself to using the previous simplification rules.

4.8 Parallel Code Generation

We describe the *Lisp parallel code generation scheme, implemented by the algorithm $PC[\![\texttt{te}]\!]\vec{\texttt{x}} = c$ which given a typed FX expression `te` and a sequence of parallel value identifiers $\vec{\texttt{x}}$, generates the parallel *Lisp code for it to run on the CM-2 processors. Parallel value identifiers in $\vec{\texttt{x}}$ are either predefined arithmetic operations, such as `+` or `-`, which are implemented on the CM-2 or user value identifiers bound by lambda abstractions.

Identifier

The compiler translates a lambda-bound identifier `x`, appearing in the sequence $\vec{\texttt{x}}$, by `x!!`. Free identifiers correspond to values that are imported in the parallel code.

$$
\begin{array}{l}
PC[\![\texttt{x}{:}\tau]\!]\vec{\texttt{x}} = \text{if } \texttt{x}{:}\tau \in \vec{\texttt{x}} \text{ then } [\![\texttt{x!!}]\!] \\
\qquad\qquad \text{else case } \tau \text{ of} \\
\qquad\qquad\qquad\qquad \begin{array}{ll}
\mathit{bool} \mid \mathit{int} \mid \mathit{real} & \Rightarrow \text{let } t = PT(\tau) \text{ in } [\![\texttt{(coerce!! x (quote } t\texttt{))}]\!] \\
\tau' \xrightarrow{\sigma} \tau'' & \Rightarrow [\![\texttt{(f-par x)}]\!] \\
\alpha & \Rightarrow [\![\texttt{(distribute x)}]\!] \\
\text{otherwise} & \Rightarrow [\![\texttt{(front-end!! x)}]\!]
\end{array}
\end{array}
$$

Compilation of Identifiers

Scalar identifiers `x` (as well as constants `1`, ... `n`) are distributed and coerced to a pvar of the corresponding type. The function PT associates a type τ with its corresponding *Lisp pvar-type. Functions identifiers `f` are translated to `(f-par f)`. Identifiers of variable type use the type dispatching expression `(distribute x)` of the FX runtime library to precisely distinguish at run time between the previous cases. Other mutable data structure identifiers are guaranteed by the static semantics to never be used.

Abstraction

In the compilation of lambda-abstractions, since *Lisp doesn't create real closures, the free pvar identifiers of the compiled lambda expression must be heap allocated, because the lambda abstraction may escape from the stack frame at which those pvar identifiers are allocated. The function ST associates a type τ with its corresponding Common-Lisp data-type.

$$
\begin{array}{l}
PC[\![\texttt{(lambda (x}{:}\tau\texttt{) te)}{:}\tau \xrightarrow{\sigma} \tau']\!]\vec{\texttt{x}} = \\
\quad \text{let } (\texttt{x}_i{:}\tau_i)_{i=1,..n} = \{\texttt{x}_i{:}\tau_i \in (\mathit{fi}[\![\texttt{te}]\!] \cap \vec{\texttt{x}}) \setminus \texttt{x}{:}\tau \mid \tau_i \text{ is not of the form } \tau_i' \xrightarrow{\sigma_i} \tau_i''\} \\
\qquad (t_i)_{i=1,..n} = ST(\tau_i) \\
\qquad (d_i)_{i=1,..n} = PTD(\texttt{x}{:}\tau_i) \\
\qquad d{=}PTD[\![\texttt{x}{:}\tau]\!] \\
\qquad c{=}PC[\![\texttt{te}]\!](\texttt{x}{:}\tau).\vec{\texttt{x}} \\
\quad \text{in } [\![\texttt{(let ((x}_1\texttt{ (}t_1\texttt{-heap-allocate x}_1\texttt{))...(x}_n\texttt{ (}t_n\texttt{-heap-allocate x}_n\texttt{)))} \\
\qquad\quad \texttt{(declare } d_1 \ldots d_n\texttt{)} \\
\qquad\quad \texttt{\#'(lambda (x!!) } d\ c\texttt{))}]\!]
\end{array}
$$

Compilation of Abstraction

Application

The parallel code generated for the application is:

$$PC[\![\texttt{(te te')}:\tau!\sigma]\!]\vec{\texttt{x}} = \text{let } c = PC[\![\texttt{te}]\!]\vec{\texttt{x}} \quad c' = PC[\![\texttt{te'}]\!]\vec{\texttt{x}} \quad \text{in } [\![\texttt{(*funcall } c\ c'\texttt{)}]\!]$$

Compilation of Application

If constructs Parallel `if` expressions are translated into *Lisp `if!!` expressions. The compilation of `if` differs from standard applications for two reasons. First, the semantics of the *Lisp expression `(if!! e e' e'')` is to execute both `e'` and `e''` and return the value of `e'` where `e` is true and `e''` otherwise. To ensure termination, we add code to check that at least one processor is active before executing the `if!!` form itself. Second, `if!!` expressions can only return pvars even though FX `if` expressions may return functions. Thus, calls to these functions must be interned within the branches of the `if` construct.

Example Before being compiled, the expression `((if b 1+ 1-) x)` is first transformed into:

```
((lambda (y) (if b (1+ y) (1- y))) x)
```

where the function expression `(if b 1+ 1-)` is abstracted over a fresh `y`. This expression gets compiled into the following *Lisp code, where `*or` performs a machine-wide reduction. If no processors are active, `nil` is returned by `*or`.

```
((lambda (y!!) (if (*or t!!)
                   (if!! b!! (1+!! y!!) (1-!! y!!))
                   (!! 0)))
               x!!)
```

■

Type Declarations

Using the information provided by type and effect inference, the compiler emits explicit *Lisp type declarations in order to improve the efficiency of the binary code produced by the *Lisp compiler.

$$ST(\tau) = \text{case } \tau \text{ of}$$
$$\begin{array}{ll} \textit{bool} & \Rightarrow [\![\texttt{boolean}]\!] \\ \textit{int} & \Rightarrow [\![\texttt{fixnum}]\!] \\ \textit{real} & \Rightarrow [\![\texttt{single-float}]\!] \\ \text{otherwise} & \Rightarrow [\![\texttt{t}]\!] \end{array}$$

$$PT(\tau) = \text{case } \tau \text{ of}$$
$$\begin{array}{ll} \textit{bool} & \Rightarrow [\![\texttt{boolean-pvar}]\!] \\ \textit{int} & \Rightarrow [\![\texttt{fixnum-pvar}]\!] \\ \textit{real} & \Rightarrow [\![\texttt{single-float-pvar}]\!] \\ \tau' \xrightarrow{\sigma} \tau'' & \Rightarrow \text{let } t = ST(\tau') \text{ and } t' = ST(\tau'') \text{ in } [\![\texttt{(function (}t\texttt{) }t'\texttt{)}]\!] \\ \text{otherwise} & \Rightarrow [\![\texttt{pvar}]\!] \end{array}$$

The figure above gives the scheme for translating type expressions into *Lisp. The function ST takes as argument a type and returns a (scalar) *Lisp type. The function PT returns a pvar-type.

$$STD[\![\texttt{x}_\tau]\!] = [\![\texttt{(declare (type } \text{case } \tau \text{ of } \textit{vector}_\rho(\tau') \Rightarrow PT(\tau') \text{ otherwise } \Rightarrow ST(\tau') \texttt{ x))}]\!]$$

The function STD gives the scalar type declarations that correspond to typed FX value identifiers.

$$PTD[\![\texttt{x}_\tau]\!] = [\![\texttt{(declare (type } PT(\tau) \texttt{ x))}]\!]$$

The function PTD gives the parallel type declarations that correspond to typed FX value identifiers.

Conclusion

We have presented a simple and effective algorithm for generating explicitly parallel and explicitly typed *Lisp programs from implicitly data-parallel and implicitly typed FX source expressions. It demonstrates that the information provided by type and effect inference can be put to work in a simple and uniform way to specify program transformations and optimizations.

Example Note that our compilation techniques support the presence of higher order functions. As an example, the parallel code generator is capable of handling the following function;

```
(lambda (f x) (lambda (y) (f x y)))
```

which partially applies `f` to `x`, by translating it into the following parallel code:

```
#'(lambda (f!! x!!)
    (declare (type pvar x!!)
             (type (function (pvar) pvar) f!!))
    (let ((x1!! (heap-allocate x!!)))
      (declare (type pvar x1!!))
      #'(lambda (y!!)
          (declare (return-pvar-p t)
                   (type pvar y!!))
          (*funcall f!! x1!! y!!)))))
```

where `x1!!` is fresh and `heap-allocate` is the generic runtime type-dispatching function for `allocate!!` on pvars. The declaration `return-pvar-p` is used to explicitly tell the *Lisp compiler that an expression returns a pvar or not ■

4.9 Optimization of Vector Operations

The code generated by the scheme described above can be dramatically improved in a variety of ways by using simple type and effect based optimizations, especially on vector operations.

A vector operation `(op e)` can be considered as a particular case of an application statement. The default mechanism for optimizing such an operation is to look at the type of the vector $vector_\rho(\tau)$ operated upon in the expression.

$SC[\![(\mathtt{op}{:}\tau\ \mathtt{te}_1 \ldots \mathtt{te}_n){:}\tau'!\sigma']\!]\vec{\tau}\vec{\mathtt{x}} =$
  let $(\mathtt{x}_i)_{i=1,..n}$ fresh
    $c_i = SC[\![\mathtt{te}_i]\!]\vec{\tau}\vec{\mathtt{x}}$ forall i in $1,..n$
    $c_{op} = SC_{\mathtt{op}}[\![\mathtt{op}{:}\tau]\!](\mathtt{x}_{1,..n})$
    $c = [\![(\mathtt{let}\ ((\mathtt{x}_1\ c_1)\ \ldots(\mathtt{x}_n\ c_n))\ c_{op})]\!]$
  in case τ' of
    $vector_\rho(\tau'') \Rightarrow$ if $\rho \in fr(\vec{\tau})$ then let $t = ST(\tau'')$ in $[\![(t\text{-}\mathtt{heap\text{-}allocate}\ c)]\!]$
                                else c
    otherwise $\Rightarrow c$

Optimization of Vector Operations

The compilation of vector operation proceeds in several steps. First, the arguments $\mathtt{te}_{1,..n}$ are translated into *Lisp code $c_{1,..n}$. Then, the code generator calls the function $SC_{\mathtt{op}}$ to translate the vector operation `op` properly, according to its type τ. Finally, a test is made to detect if the value returned by the operation `op` is a vector, and if that vector shall be allocated in the heap, according to the criterion 4.1. This is done by checking that the type τ' of the operation is of the form $vector_\rho(\tau'')$ and that ρ occurs in the sequence $\vec{\tau}$.

$SC_{\texttt{op}}[\![\texttt{op}{:}\tau]\!]\texttt{x}_{1,..n} =$
case τ of
$\tau_1 \times vector_\rho(\tau_2) \xrightarrow{\sigma} \tau_3 \Rightarrow$ let $t = ST(\tau_2)$
$c_1 = [\![$`(with-context-of x2 (t-op x1,..n))`$]\!]$
$c_2 = [\![$`(front-end-op x1,..n)`$]\!]$
in if $PF(\tau_1)$ then $[\![$`(if (f-struct-p x1) c1 c2)`$]\!]$
else c_2
$vector_\rho(\tau_1) \times \tau_2 \xrightarrow{\sigma} \tau_3 \Rightarrow$ let $t = ST(\tau_2)$ in $[\![$`(with-context-of x1 (t-op x1,..n))`$]\!]$
$\tau_1 \xrightarrow{\sigma} vector_\rho(\tau_2) \Rightarrow$ let $t = ST(\tau_2)$ in $[\![$`(with-context x1 (t-op x1,..n))`$]\!]$
otherwise $\Rightarrow [\![$`(op x1,..n)`$]\!]$

Translation of Vector Operations

When τ is a scalar data type, such as *bool*, *int* or *real*, the vector is implemented by an unboxed pvar of scalar components. In this case, $SC_{\texttt{op}}$ translates the operation by a call to the appropriate *Lisp macro, defined in the runtime library with the name t-`op`, where t is the *Lisp translation of τ and `op` stands for the name of the vector operation.

When τ is a non-scalar data type, such as a *list* or a *vector*, the vector is implemented as a boxed front-end pvar, and the operation `op` by the macro `front-end-op` of the runtime library. Otherwise, τ is a type variable and the compiler translates the operation into the call to the generic function of the runtime library implementing the operation, named `op`.

Example Type information can be used to optimize the compilation of higher-order functions, such as the vector operation `vector-map`. For instance, assume that the mapped vector has integer elements, as in the example below.

```
(lambda (f x) (vector-map (if true 1+ f) x))
```

The compiled code invokes the macro `int-vector-map` and expands it into:

```
(lambda (f x)
  (let ((x1 (if t (make-f-struct #'1+ #'1+!!) f)))
    (if (f-struct-p x1)
      (int-vector-map x1 x)
      (front-end-vector-map x1 x)
```

The first operation performed in this code is to bind with the fresh identifier `x1` to the f-structure which corresponds to the function expression `(if true 1+ f)`. Because `1+` is a parallelizable predefined function, an f-structure is created for it by `(make-f-struct #'1+ #'1+!!)`. Then, the operation `(f-struct-p f)` checks that `x1` is a f-structure itself. In this case, the mapping operation is translated by invoking the appropriate macro `int-vector-map` that implements the parallel map of a function on integers.

```
(with-context-of x
  (coerce!! (*funcall (f-par x1) x) 'fixnum-pvar))
```

It performs a parallel call, using `*funcall`, to the parallel implementation `(f-par x1)` of `x1`, the result of which the result coerced back into a `fixnum-pvar` ■

Otherwise, the operation is performed sequentially by using the function `front-end-vector-map`. In this code, no runtime type test on `x` is necessary, since type checking guarantees that `x` is a vector of integers.

Example In a way similar to types, effects can be used to specialize some higher-order vector operations which require runtime tests. To describe the optimization performed at such program points, let us consider the case of compiling a form such as:

```
(lambda (x) (vector-map 1+ x))
```

Since the effect of `1+` is $\emptyset$ and its arguments type *int* scalar, the compiled code is:

```
(lambda (x)
  (with-context-of x
    (coerce!! (1+!! x) 'fixnum-pvar)))
```

Note that in the code above, it is not tested by using `f-struct-p` that `(make-f-struct #'1+ #'1+!!)` is an f-structure. There is no boxing and unboxing of the form `(f-par (make-f-struct #'1+ #'1+!!))` and there is no `*funcall` to `#1+!!`, because the simplification rules of the previous section apply ■

On the contrary, if the higher-order function given to `vector-map` has some latent side-effect or if the type τ of the vector elements it manipulates are not scalar, the operation `vector-map` is compiled by its sequential implementation, `front-end-vector-map`, bypassing the runtime tests present in the runtime library.

4.10 Compilation of let

Managing the `let` construct (and, similarly, `letrec`) within the simplified compilation scheme shown above is easy. First, we have to distinguish between value and function definitions:

Definition 4.3 (Function Identifier) *An identifier `f` is defined as a function when it is bound in a `let` expression to an explicit lambda expression. Otherwise, it is considered as a value definition.*

Value definitions such as `(let ((x e)) e')` are handled according to our basic compilation scheme by translating them into `((lambda (x) e') e)`.

Function Definitions

Function definitions are translated by using the Common-Lisp `labels` construct. When `(lambda (x) e)`, bound to `f` in a `let` construct is a function that can be parallelized, its parallel implementation is associated with `f!!`.

A finite map m replaces the sequence of parallelized functions $\vec{\mathtt{x}}$. In m, the identifiers `f` of parallelizable function are associated with the sequence of lambda-bound function identifiers $\vec{\mathtt{f}}$ on which the parallelizability of `f` actually depends at runtime.

$$
\begin{array}{l}
SC[\![\texttt{(let (f (lambda (x:}\tau_1\texttt{) te):}\tau_1 \xrightarrow{\sigma_1} \tau_2\texttt{) te'):}\tau_3!\sigma_2]\!]\vec{\tau}, m = \\
\quad \text{let } d = STD[\![\texttt{x:}\tau_1]\!] \\
\qquad c_1' = SC[\![\texttt{te}]\!]\vec{\tau}, m \\
\qquad c_1 = [\![\texttt{(lambda (x) } d\ c_1'\texttt{)}]\!] \\
\quad \text{in if } PF(\tau_2) \\
\qquad \text{then let } \vec{\mathtt{f}} = \{\texttt{f:}\tau_f \overset{\varsigma_f \cup \sigma_f}{\rightarrow} \tau_f' \in \mathit{fi}[\![\texttt{te}]\!] \setminus \texttt{x:}\tau_1 \mid \varsigma_f \in \sigma_1\} \\
\qquad\qquad d_2 = PTD[\![\texttt{x:}\tau_1]\!] \\
\qquad\qquad c_2' = PC[\![\texttt{te}]\!]\texttt{x:}\tau_1 \\
\qquad\qquad c_2 = [\![\texttt{(lambda (x!!) } d_2\ c_2'\texttt{)}]\!] \\
\qquad\qquad c_3 = SC[\![\texttt{te'}]\!](\tau_3.\vec{\tau}), (m + \{\texttt{f} \mapsto \vec{\mathtt{f}}\}) \\
\qquad\quad \text{in } [\![\texttt{(labels ((f } c_1\texttt{)(f!! } c_2\texttt{)) } c_3\texttt{)}]\!] \\
\qquad \text{else let } c_2 = SC[\![\texttt{te'}]\!](\tau_3.\vec{\tau}), m \\
\qquad\quad \text{in } [\![\texttt{(let ((f \#'}c_1\texttt{)) } c_2\texttt{)}]\!]
\end{array}
$$

Compilation of Definitions

Function Identifiers

An occurrence of a function identifier `f` in function position `(f e)` is left as such during sequential code compilation. An occurrence in value position, such as in `(lambda (x) f)` or `(e f)` is translated, as previously, by the allocation of an f-structure if `f` was added to $\vec{\mathtt{x}}$.

$$
\begin{array}{l}
SC[\![\mathtt{x}{:}\tau]\!]\vec{r}, m = \\
\quad \text{if } \mathtt{x} \notin Dom(m) \text{ then } [\![\mathtt{x}]\!] \\
\qquad\qquad \text{else let } \mathtt{x}_{1,..n} = m(\mathtt{x}) \\
\qquad\qquad\quad \text{in } [\![\mathtt{(if\ (and\ (f\text{-}struct\text{-}p\ x}_1\mathtt{)}\ldots\mathtt{(f\text{-}struct\text{-}p\ x}_n\mathtt{))} \\
\qquad\qquad\qquad\quad \mathtt{(make\text{-}f\text{-}struct\ \#'x\ \#'x!!)} \\
\qquad\qquad\qquad\quad \mathtt{x)}\ ]\!]
\end{array}
$$

Compilation of Identifiers

If `f` is parallelizable (modulo the possible run-time check on its free function identifiers) the f-structure `(make-f-struct #'f #'f!!))` is returned. Otherwise, `#'f` is generated.

Example As an example, the following definition:

```
(lambda (g)
    (let ((f (lambda (x)
                (g x))))
        f))
```

is compiled as:

```
(lambda (g)
   (labels (((f x)
               (funcall (if (f-struct-p g) (f-seq g) g)
                        x))
            ((f!! x!!)
               (*funcall (f-par g) x!!)))
      (if (f-struct-p g)
          (make-f-struct #'f #'f!!)
          #'f)))
```

During parallel code generation, an occurrence of a parallelizable function identifier `f` in function position is translated to `f!!`. The static semantics guarantees that non-parallelizable functions occurring in parallelizable FX expressions are never used. The occurrence of `f` in a value position, such as in `(lambda (x) f)` or `(e f)`, is translated into #'`f!!` ■

4.11 Implementation

The FX compiler for the CM-2 was implemented by using the initial implementation of the FX-91 interpreter, consisting of a parser, a kind and type checker and a simple interpreter. The FX-91 interpreter is written in Scheme and runs under T [Rees & al., 1984] and has been adapted to *Lisp. The techniques described in this chapter have all been implemented. The examples given in this chapter have all been run using this implementation.

The FX/CM Compiler has first been tested on the *Lisp simulator [*Lisp, 1987]. We have used the CM-2 installed at ETCA (Arcueil, France) and run other data-parallel algorithms, such as a `life` program, a `quicksort` algorithm using segmented scanning and a matrix transposition algorithm.

Example We illustrate the speed-up obtained by our compiler by the case of the segmented matrix transposition algorithm `segment-transpose` (described below) which uses higher-order functions and segmented vectors as 2D matrices.

```
(define (segment-transpose m segment)
  (let ((id (segment-identity segment))
        (offset (1+ (vector-reduce max id))))
    (permute m
     (vector-map2 + (segment-index segment)
                  (vector-map (lambda (i) (* offset i))
                              id)))))
```

It was executed on 4×4 to 128×128 integer matrices on a 8K processor CM-200a. Busy times are almost independent of the problem size, except for the last case. For a 128×128 matrix, the VP-set exceeds the actual machine size and thus multiple operations are performed on each processor, slowing down the overall execution ■

Matrix size	8×8	16×16	32×32	64×64	128×128
Elapsed Time (s)	1.48×10^{-2}	1.51×10^{-2}	1.83×10^{-2}	1.52×10^{-2}	1.74×10^{-2}
Busy Time (s)	6.47×10^{-3}	6.48×10^{-3}	6.43×10^{-3}	6.43×10^{-3}	1.04×10^{-2}

4.12 Related Work

In order to go beyond the *Lisp system [*Lisp, 1987], several other programming paradigms have been suggested. The major proposals are the APL-inspired alpha and beta global operations on xectors in CM-Lisp [Hillis, 1985], the paralation abstract data type and its element-wise operations in Paralation-Lisp [Sabot, 1990] and the scan operations over segmented vectors in SV-Lisp [Blelloch, 1990].

In the *Lisp language, the low-level programming features that are introduced in order to efficiently use the CM are all easy to compile. Not so for these other languages. However, their reference manuals [Hillis, 1985, Sabot, 1990, Blelloch, 1990] are quite elusive on the issue of which compile-time analysis would be necessary for programs to be effectively compiled.

Nonetheless, the data-parallel constructs designed in the SV-Lisp language [Blelloch, 1990] or in the V-Code [Blelloch & Chatterjee, 1990] provide a programming model that made its way into the design of the vector module for the FX language. It proved here to be effectively compilable using our type and effect system. Our approach is thus not to introduce a new data model, but to describe how a sophisticated static system can be used to safely implement data parallel constructs on a massively parallel machine in the presence of side effects.

4.13 Conclusion

The FX/CM compiler prototype supports the claim that effect systems can be used to integrate, for the first time, functional and imperative programming on massively parallel architectures. Effects are used to decide whether potentially parallel constructs can actually be implemented as such without leading to non-determinism. Regions are used to optimize space allocation strategies and limit the garbage collection overhead. We expect to extend our system to multi-dimensional arrays, using ideas expressed in [Blelloch, 1990, Sabot, 1990]. We are also looking at other applications of regions, for instance using them to manage the alignment of vectors to minimize communication costs.

Chapter 5

Future Work

In [Tofte & Talpin, 1993], we propose an application of the type and effect inference system of chapter 2 for imposing a stack-based allocation discipline to functional programs. It consists of associating a region with the type of every data structure manipulated in a program and then determining the lexical scope of regions by using an observation criterion.

Translation Process

The translation process described in the next figure associates every source expression `e` with an expression `e'` annotated so as to explicitly specify the scope, instantiation and generalization of data regions associated with the value of expressions.

The directive `(r-let (r) e)` locally defines a region in the expression `e` that can be collected when the execution of `e` terminates. The directive `(r-abs (r) e)` abstracts the expression `e` over the region `r` which can be applied to a region `r'` by `(r-app (r') e)`

$$\frac{\mathcal{E}(\mathtt{x}) = \tau}{\mathcal{E} \vdash \mathtt{x} \Rightarrow \mathtt{x} : \tau, \emptyset} \qquad \frac{\mathcal{E}(\mathtt{x}) = \forall \vec{\alpha}\vec{\varrho}\vec{\varsigma}.\tau \qquad \theta = \{\vec{\alpha} \mapsto \vec{\tau}, \vec{\varrho} \mapsto \vec{\rho}, \vec{\varsigma} \mapsto \vec{\sigma}\}}{\mathcal{E} \vdash \mathtt{x} \Rightarrow \mathtt{(r\text{-}app\ (}\vec{\rho}\mathtt{)\ x)} : \theta\tau, \emptyset} \quad \text{(t-var)}$$

$$\frac{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} \Rightarrow \mathtt{e}' : \tau', \sigma}{\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} \Rightarrow \mathtt{(lambda\ (x)\ e')} : \tau \xrightarrow{\sigma \cup \sigma'} \tau', \emptyset} \quad \text{(t-abs)}$$

$$\frac{\mathcal{E} \vdash \mathtt{e}_1 \Rightarrow \mathtt{e}_1' : \tau \xrightarrow{\sigma} \tau', \sigma' \qquad \mathcal{E} \vdash \mathtt{e}_2 \Rightarrow \mathtt{e}_2' : \tau, \sigma''}{\mathcal{E} \vdash \mathtt{(e_1\ e_2)} \Rightarrow \mathtt{(e_1'\ e_2')} : \tau', \sigma \cup \sigma' \cup \sigma''} \quad \text{(t-app)}$$

$$\frac{exp[\![\mathtt{e}]\!] \qquad \mathcal{E} \vdash \mathtt{e}_1 \Rightarrow \mathtt{e}_1' : \tau, \sigma \qquad \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e}_2 \Rightarrow \mathtt{e}_2' : \tau', \sigma'}{\mathcal{E} \vdash \mathtt{(let\ (x\ e_1)\ e_2)} \Rightarrow \mathtt{(let\ (x\ e_1')\ e_2')} : \tau', \sigma \cup \sigma'} \quad \text{(t-ilet)}$$

$$\frac{\neg exp[\![\mathtt{e}]\!] \quad \mathcal{E} \vdash \mathtt{e}_1 \Rightarrow \mathtt{e}_1' : \tau, \sigma \quad \forall \vec{\alpha}\vec{\varrho}\vec{\varsigma}.\tau = Gen(\mathcal{E})(\tau) \quad \mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \forall \vec{\alpha}\vec{\varrho}\vec{\varsigma}.\tau\} \vdash \mathtt{e}_2 \Rightarrow \mathtt{e}_2' : \tau', \sigma'}{\mathcal{E} \vdash \mathtt{(let\ (x\ e_1)\ e_2)} \Rightarrow \mathtt{(let\ (x\ (r\text{-}abs\ (}\vec{\varrho}\mathtt{)\ e_1'))\ e_2')} : \tau', \sigma \cup \sigma'} \quad \text{(t-let)}$$

$$\frac{\mathcal{E} \vdash \mathtt{e} \Rightarrow \mathtt{e}' : \tau, \sigma \qquad \sigma' = Observe(\mathcal{E}, \tau)(\sigma) \qquad \vec{\rho} = fr(\sigma \setminus \sigma')}{\mathcal{E} \vdash \mathtt{e} \Rightarrow \mathtt{(r\text{-}let\ (}\vec{\rho}\mathtt{)\ e')} : \tau', Observe(\mathcal{E}, \tau')(\sigma \cup \sigma')} \quad \text{(t-obs)}$$

Annotating Expressions with Data Regions

The generalization of the value of expressions using the directive `r-abs` is performed by the lexical binding of value identifiers to the value of non expansive expressions, which are usually functions. The instantiation of data regions by the directive `r-app` is performed for every occurrence of these bound identifiers. Finally, the rule (obs) of observation is used to lexically isolate non observable data regions $\vec{\rho}$.

Example The process of translating source expressions with explicit annotations about data regions is illustrated by this last figure related to the definition of the function `fold` previously studied in section 3.7.

```
(lambda (f i)
  (lambda (l)
    (r-let (r1)
      (let (result ((r-app (r1) new) i))
        (r-let (r2)
          (let (data ((r-app (r2) new) l))
            (until (null? ((r-app (r2) get) data))
              (((r-app (r1) set) result) (f (car ((r-app (r2) get) data))
                                            ((r-app (r1) get) result)))
              (((r-app (r2) set) data) (cdr ((r-app (r2) get) data))))))
        ((r-app (r1) get) result)))))
```

Note that the region of `data`, noted `r2`, is not mentioned outside the `let` construct of its definition. The region `r1` is also local to the function.

Polymorphic Recursion

One of the major unflexibilities in the Hindley/Milner type system is that recursive functions remains monomorphic in the body of their definition.

$$\frac{\mathcal{E}_{\mathtt{f}} + \{\mathtt{f} \mapsto \tau\} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset}{\mathcal{E} \vdash \mathtt{(rec\ (f\ x)\ e)} : \tau, \emptyset} \qquad \text{(rec)}$$

The impact of this feature on our storage optimization scheme will be to force recursive invocations of a function to occasionally refer to the same data regions.

Example This is significant in the following version of the factorial procedure that uses pointers to store its intermediate results.

```
(define fact (lambda (n r)
               (if (<= n 1) ((set r) 1)
                 (let (i (new n))
                   (fact (- n 1) i)
                   ((set r) (* n (get i)))))))
```

Without polymorphism inside the recursive function definitions, the function `fact` is assigned type:

$$\mathtt{fact} : int \times ref_{\rho}(int) \xrightarrow{\varsigma \cup init(\rho, int) \cup read(\rho) \cup write(\rho)} unit$$

This means that ρ is the region of every successive temporary pointer `i`. This result is evident in the following output code, where `r1` is abstracted over the whole `lambda` abstraction global region:

```
(lambda (n r)
  (if (<= n 1) (((r-app (r1) set) r) 1)
    (let (i ((r-app (r1) new) n))
      (fact (- n 1) i)
      (((r-app (r1) set) r) (* n ((r-app (r1) get) i))))))
```

A solution to the general problem of regions whose sizes cannot be computed at compile-time is to implement them by a linked list of pages, each page having a fixed size. When a page is full, the memory manager allocates a new page and links it to the previous one. Deallocation of a region then just puts the region into a "free-list" of available pages. Under this scheme, the runtime system works like a generational garbage collection scheme, where entire regions are collected at a time.

The problem of "polymorphic recursion" is to find the polymorphic type of a recursive function which exactly satisfies the following recursive definition:

$$\frac{\mathcal{E}_{\mathtt{f}} + \{\mathtt{f} \mapsto Gen(\mathcal{E})(\tau)\} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset}{\mathcal{E} \vdash \mathtt{(rec\ (f\ x)\ e)} : \tau, \emptyset} \qquad \text{(poly-rec)}$$

But this problem has been proved reducible to semi-unification [Henglein, 1992], which is unfortunately undecidable [Kfoury & Tiuryn, 1990]. Nonetheless, decidable restrictions to "uniform" cases have been investigated [Leiss & Henglein, 1991]. We expect to show in [Tofte & Talpin, 1993] that the restriction of the problem of polymorphic recursion to the inference of polymorphic region in recursive functions is decidable.

$$\frac{\mathcal{E}_{\mathtt{f}} + \{\mathtt{f} \mapsto \forall\vec{\rho}.\tau\} \vdash \mathtt{(lambda\ (x)\ e)} : \tau, \emptyset \qquad \vec{\varrho} = frv(\tau) \setminus frv(\mathcal{E})}{\mathcal{E} \vdash \mathtt{(rec\ (f\ x)\ e)} : \tau, \emptyset} \qquad \text{(gen-rec)}$$

One of the main trends for extending type and effect inference is certainly to improve the inference of regions in recursive functions in this way. This improvement has indeed a dramatic impact on optimization that can be performed by our type and effect inference system.

Example In the previous example, using the rule (gen-rec) permits allocation of the pointer `i`, which store the intermediate results of the function `fact`, in separate regions `r2`, which are lexically isolated by a `r-let` construct and do not thus survive the recursive call of the function.

```
(r-abs (r1) (lambda (n r)
              (if (<= n 1) (((r-app (r1) set) r) 1)
                (r-let (r2)
                  (let (i ((r-app (r2) new) n))
                  ((r-app (r2) fact) (- n 1) i)
                  (((r-app (r1) set) r) (* n ((r-app (r2) get) i)))))))))
```

Chapter 6

Conclusion

Contribution

We introduced a new type system to reconstruct the principal type, region and effect of expressions introducing a tantamount notion to subtyping in the domain of effects, *subeffecting*. We have extended the principle of polymorphic type inference of [Milner, 1978] to the reconstruction of regions and effects. Regions statically describe uniform sharing relations between values and, incidentally, delimit the scope-locality of data.

We have formally developed both the practical and theoretical consequences of this observation. On the practical side, static information such as the type, region and effect of expressions assists in the design and implementation of compile-time techniques for optimizing the management of data.

Combining types and effects allows the integration of imperative programming features with polymorphic functional languages. We introduced a typing discipline using effect inference to determine the principal type of expressions in the presence of assignment - the type and effect discipline.

In our typing discipline, typing references is done by inferring allocation effects which tells the data type pointed at by regions of initialized references. Effects are used to control type generalization in the presence of imperative constructs. To type a `let` construct, the allocation effect of the bound expression provides all the necessary information to determine which type variables must not be generalized.

By using an observation criterion, our typing discipline limits the report of effects to those that affect accessible regions. The observable effects of an expression range over the regions that are free in its type environment and its type. Effects related to local data structures can be discarded during type reconstruction. The type of an expression can be generalized with respect to the variables that are not free in the type environment or in its observable effect.

The notion of observable effects is crucial to distinguishing the functions which only use references locally and implement their purely functional counterpart with a more imperative style. By using effect information together with an observation criterion, our type system is able to precisely delimit the scope of side-effecting operations, thus allowing type generalization to be performed in `let` expressions in a more efficient and uniform way than previous type systems.

Concluding Remarks

Among other formal methods, static typing is the most widely used technique of static analysis in programming languages and is also one of the most popular. Static typing consists of detecting the most frequent cause of errors occurring during the execution of a program, the inconsistent use of a data structure. Static type checking detects algorithmic errors in programs by verifying type declarations. Some of the efforts needed to correctly specify types in programs can be spared by using a type inference engine, which permits omitted declarations to be automatically reconstructed.

Because a type system must be simple for the user to understand it and because users may nonetheless write complex programs, any type system affects the expressive power of the language it is designed for by

forcing it to reject some programs which are correct but cannot be recognized as such.

The tension is the most serious in the case of functional languages which are supposed, at the very least, to support generic functions. In contrast to dynamic typing, which appears to be the simplest solution for the implementation of such languages, introducing the notion of type polymorphism allows generic functions to be expressed and has definitely contributed to making functional programming more popular.

By using type inference, functional languages still possess some unflexibilities of use which can be somewhat circumvented by using explicit polymorphic typing. Explicit polymorphic typing provides all the expressive power that the user may need to mitigate any lack of expressiveness originating from a weakness of the type system used.

Functional languages are remarkably well-designed in that they are based on sound theoretical foundations. Imperative features are nonetheless required to make programming such languages feasible. However, integrating polymorphic typing in an imperative language appears problematic. This integration shall not compromise the correctness of the language's semantics.

By using effect systems, imperative features can be integrated in functional languages and easily comprehended by users according to strong theoretical bases. Resolving the tension between imperative programming and type polymorphism is effectively solved by using an effect system. An effect system provides the appropriate medium to denote state transitions at the level of the type system in programming languages semantics.

Just as types represent what programs compute, effects denote how programs compute. Regions appear in types and effects, and represent precise sharing relations between data. As a result, type, region and effect inference establishes a very strong relationship between structural information: types, relational information: regions, behavioral information: effects. For the user, it offers precise, general and comprehensible documentation about programs.

For the implementor, this relationship eases the specification of optimization techniques based on semantic information and provides general and efficient static analysis techniques for the implementation of functional programming languages. It allows better code to be generated and data to be represented efficiently.

When functional and imperative programming paradigms are integrated together within a programming language, implementors must exert care when designing code optimizers, since side effects inhibit most of the nice properties of pure functional languages which are put to work in code transformations.

When going from sequentiality to parallelism the issues get significantly more complicated at the programmer and implementor levels. Concomitant use of side effects and parallelism leads to nondeterminism, which makes program understanding and debugging difficult because of the non-reproductibility of results. Restricting parallel programs to be deterministic is a way of making parallel program design in higher-order imperative languages a more manageable task.

The FX compiler prototype for the CM-2 enforces such deterministic constraints. It supports the claim that effect systems can be used to integrate functional and imperative styles for the implementation of programming languages on massively parallel architectures. Effects are used to decide whether implicitly data-parallel vector operations can actually be implemented on a parallel architecture without leading to non-determinism. Regions are used to optimize the strategies of space allocation.

The initial design goal of effect systems was to safely integrate functional and imperative paradigms. The type and effect discipline permits the imperative programming style to be fully integrated in a functional language equipped with a polymorphic type system. Using type, region and effect inference, we demonstrated how an analysis technique and its use could be considered within the same framework.

Bibliography

[Abadi & al., 1992] Abadi, M., Cardelli, L., Pierce. B., and Rémy, D. Dynamic typing in polymorphic languages. In *Proceedings of the 1992 workshop on ML and its applications*, 1992.

[Abelson & Sussman, 1985] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[Abramsky, 1986] S. Abramsky. Strictness analysis and polymorphic invariance. In *Proc. Workshop on Programs as Data Objects*, pages 1-23, Berlin, 1986. Springer LNCS 217.

[Abramsky & Jensen, 1991] S. Abramsky and T.P. Jensen. A relational approach to strictness analysis for higher-order polymorphic functions. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 49-54, 1991.

[Aczel, 1988] Aczel, P. *Non well-founded sets*. CSLI Lectures Notes, Stanford, 1988.

[Aho & al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[Appel & MacQueen, 1987] Appel, A. W., and MacQueen, D. B. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture 1987*, volume 242 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[Appel & Mac Queen, 1990] Appel, A. W., and Mac Queen, D. B. Standard ML Reference Manual. AT&T Bell Laboratories and Princeton University, October 1990.

[Appel, 1992] Appel, A. W. *Compiling with continuations*. Cambridge University Press, 1992.

[Baker, 1990] Baker, H. G. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Lisp and Functional Programming 1990*, pages 218-226. ACM Press, 1990.

[Barendregt, 1984] Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics*, second edition. North-Holland Publishing, Amsterdam, 1984.

[Berthomieu, 1989] Berthomieu, B. Implementing CCS: the LCS experiment. Technical report 89425, LAAS, December 1989.

[Blelloch, 1990] Blelloch, G. E. Vector model for data-parallel computing. The MIT Press, Cambridge, 1990.

[Blelloch & Chatterjee, 1990] Blelloch, G. E., and Chatterjee S. VCODE, A Data-Parallel Intermediate Language. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computers*. Journal of Parallel and Distributed Computing, February 1990.

[Blelloch & Sabot, 1990] Blelloch, G. E., and Sabot, G. W. Compiling collection oriented languages onto massively parallel computers. In *Journal of Parallel and Distributed Computing*, volume 8, pages 119-134. Academic Press, 1990.

[Blelloch & Little, 1988] Blelloch, G. E., and Little, J. J. Parallel solutions to geometric problems on the scan model of computations. *AI Laboratory Memo No. 952*. Massachusetts Institute of Technology, February 1988.

[Cardone & Coppo, 1991] Cardone, F., Coppo, M. Type Inference with Recursive Types: Syntax and Semantics. In *Information and Computation*, Vol. 92, pages 48-80. Academic Press 1991.

[Chirimar & al., 1992] Chirimar, J., Gunter, C. A., and Riecke, J. G. Proving memory management invariants for a language based on linear logic. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 151-160. ACM, New-York, 1992.

[Clément & al., 1985] Clément, D., Despeyroux, J., Despeyroux, T., and Kahn, G. A simple applicative language: Mini-ML. Technical Report 529, INRIA, 1986.

[Corbin & Bidoit, 1983] Corbin, Bidoit A Rehabilitation of Robinson's Unification Algorithm. In *IFIP'83*. North-Holland, 1983.

[Coquand & Huet, 1988] Coquand, T., and Huet, G. The calculus of constructions. *Information and Computation*, 76, 1988.

[Cousineau & al., 1987] Cousineau, G., Curien, P. L. and Mauny, M. The categorical abstract machine. *Science of Computer Programming*, 8(2):173-202, 1987.

[Cousineau & Huet, 1987] Cousineau, G., and Huet, G. The CAML primer. Technical report 122, INRIA, 1990.

[Cousot & Cousot, 1977] Cousot, P., and Cousot, R. Abstract Interpretation, a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the 1977 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1977.

[Cousot & Cousot, 1992] Cousot, P., and Cousot, R. Inductive definitions, semantics and abstract interpretation. In *Principles of Programming Languages 1992*, pages 83-94. ACM Press, 1992.

[Cousineau & al., 1987] Guy Cousineau, G., Curien, P.-L., and Mauny, M. The Categorical Abstract Machine. In *Science of Computer Programming*, volume 8, number 2, pages 173-202, 1987.

[Damas, 1985] Damas, L. Type Assignment in Programming Languages. *Ph.D. Thesis*, University of Edinburgh, April 1985.

[Damas & Milner, 1982] Damas, L. and Milner, R. Principal type-schemes for functional programs. In *Principles of Programming Languages 1982*, pages 207-212. ACM Press, 1982.

[Deutsch, 1990] Deutsch A. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-Order Functional Specifications. In *Proceedings of the 1990 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1990.

[Deutsch, April 1992] Deutsch A. A Storeless Model of Aliasing and its Abstractions using Finite Representations of Right-Regular Equivalence Relations. In *Proc. of the IEEE 1992 International Conference on Computer Languages*, pages 2-13. IEEE Press, Oakland, April 1992.

[Deutsch, 1992] Deutsch A. Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data. *Ph.D. Thesis*, Université Paris VI, 1992.

[Dijkstra, 1960] Dijkstra, E. W. Recursive programming. In *Numerische Math*, volume 2, pages 312–318, 1960.

[Duba & al., 1991] Duba B. F., Harper R., and MacQueen D. Typing First-Class Continuations in ML. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1991.

[Felleisen & Friedman, 1989] Felleisen, M., and Friedman, D. P. A syntactic theory of sequential state. In *Theoretical Computer Science*, volume 69, number 3, pages 243-287, 1989.

[Fortran90] ANSI, *American National Standard for Information Systems Programming Language Fortran: S8(X3.9-198x)*, March 1989.

[Gallier, 1991] Gallier, G. Constructive Logic, Part I: a tutorial on proof systems and typed lambda calculi. *Technical Report No. 8.* DEC Paris Research Laboratory, May 1991.

[Giannini & Della Rocca, 1988] Giannini, P., and Della Rocca, S. R. Characterization of typing in polymorphic type disciplines. In *the proceedings of the 1988's IEEE Conference on Logic in Computer Science*, pages 61-70. IEEE Press, Oakland, June 1988.

[Gifford & al., 1987] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. FX-87 Reference Manual. *MIT/LCS/TR-407*, MIT Laboratory for Computer Science, September 1987.

[Girard, 1972] Girard, J. Y. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'État, Université Paris VII, 1972.

[Girard, 1986] Girard, J. Y. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 2(45):159-192, 1986.

[Girard, 1987] Girard., J.-Y. Linear logic. *Theoretical Computer Science*, volume 50, pages 1-102, 1987.

[Goldberg & Park, 1990] Goldberg. B., and Park, Y. G. Higher order escape analysis, optimizing stack allocation in functional program implementation. In *the proceedings of the 1990's European Symposium on Programming*, volume 432 of the *Lectures Notes in Computer Science*, pages 152-160. Springer Verlag, 1990.

[Goldberg, 1991] Goldberg. B. Tag-free garbage collection for strongly typed programming languages. In *SIGPLAN conference on Programming Language Design and Implementation*, 1991.

[Goldberg, 1992] Goldberg. B. Polymorphic type reconstruction for garbage collection without tags. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 53-65. ACM Press, New-York, 1992.

[Gordon & al., 1979] Gordon, M. J., Milner, A. J., and Wadsworth, C. P. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Guzman & Hudak, 1990] Guzman, J.C., and Hudak, P. Single-Threaded Polymorphic Lambda Calculus. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. June 1990.

[Hendren, 1990] Hendren, L. Parallelizing programs with recursive data structures. In *IEEE transactions on Parallel and Distributed Processing*, volume 1, pages 35-47. January 1990.

[Hammel & Gifford, 1988] Hammel, R. T., and Gifford, D. K. FX-87 Performance Measurements: Dataflow Implementation. *MIT/LCS/TR-421*, MIT Laboratory for Computer Science, November 1988.

[Van Leuven, ed., 1990] Van Leeuwen, J., editor. *Handbook of Theoretical Computer Science, volume B.* The MIT Press/Elsevier, 1990.

[Hannan, 1990] Hannan, J. Investigating a Proof-Theoretic Meta-Language for Functional Languages. *Ph. D. Thesis*, University of Pennsylvania, 1990.

[Hannan & Pfenning, 1992] Hannan, J., and Pfenning, F. Compiler verification in LF. In *the proceedings of the 1992's IEEE Conference on Logic in Computer Science*. Santa Cruz, California, June 1992.

[Harrison, 1989] Harrison, W. L. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. In *Lisp and Symbolic Computation, an Internal Journal*, 2 (3). 1989.

[Harper & al., 1987] HARPER, R., HONZEL, F., AND PLOTKIN, G. A framework for defining logics. *Technical Report CMU-CS-89-173*, School of Computer Science, Carnegie Mellon University, 1989.

[Harper & Lillibridge, 1992] HARPER, R., AND LILLIBRIDGE, M. Polymorphic type assignment and CPS conversion. In *1992 SIGPLAN Continuations Workshop*, 1992.

[Harper & Lillibridge, 1993] HARPER, R., AND LILLIBRIDGE, M. Explicit polymorphism and CPS conversion. In *Principles of Programming Languages 1993*. ACM Press, 1993.

[Henglein, 1992] HENGLEIN, F. Type inference with polymorphic recursion. In *ACM Transactions on Programming Languages and Systems*. ACM Press, 1992.

[Hillis, 1985] HILLIS, W. D. The Connection Machine. The MIT Press, Cambridge, 1985.

[Hindley, 1969] HINDLEY, R. The principal type scheme of an object in combinatory logic. In *Transaction of the American Mathematical Society*, volume 146, pages 26-60, 1969.

[Hoang & al., 1992] HOANG, M., MITCHELL, J., AND VISWANATHAN, R. Standard ML weak polymorphism and imperative constructs. Unpublished. Accessible by anonymous ftp on `theory.stanford.edu`. Stanford University, December 1992.

[Hoare, 1985] HOARE, C. A. R. *Communicating sequential processes*. Prentice-Hall, 1985.

[Hudak, 1986] HUDAK, P. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on Programming Language Design and Implementation*. ACM, New-York, August 1986.

[Hudak & al., 1992] HUDAK, P., AND AL. Report on the programming language Haskell. *SIGPLAN Notices*, volume 27(5), section R. ACM Press, 1992.

[Huelsbergen & Larus, 1992] HUELSBERGEN, L., AND LARUS, J. R. Dynamic program parallelization. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 311-323. ACM Press, New-York, 1992.

[Huet, 1976] HUET, G. Résolution d'équations dans les langages 1,2,... ω. *Doctoral dissertation*. Université Paris VII, 1976.

[Huet, 1989] HUET, G. The Calculus of Constructions, documentation and user's guide. Technical report 110, INRIA, 1989.

[Hughes, 1987] HUGHES J. Backward Analysis of Functional Programs. In *Proceedings of the Workshop on Partial Evaluation and Mixed Computation*. North Holland, October 1987.

[Jouvelot & Gifford, 1989] JOUVELOT, P., AND GIFFORD, D. K. Reasoning about Continuations with Control Effects. *MIT/LCS/TR-378*, MIT Laboratory for Computer Science, 1989.

[Jouvelot & Gifford, 1991] JOUVELOT, P., AND GIFFORD, D. K. Algebraic reconstruction of types and effects. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1991.

[Kahn, 1988] KAHN, G. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237-257. Elsevier, 1988.

[Kanellakis, 1989] KANELLAKIS, P., AND MITCHELL, J. C. Polymorphic unification and ML typing. In *Principles of Programming Languages 1989*. ACM Press, 1989.

[Kanellakis, 1991] KANELLAKIS, P.C., MAIRSON, H.G. AND MITCHELL, J.C. Unification and ML type reconstruction. In *Computational Logic, Essays in Honor of Alan Robinson,* ed. J.-L. Lassez and G.D. Plotkin, pages 444-478. The MIT Press, 1991.

[Kapur & al., 1989] D. KAPUR, D. MUSSER, P. NARENDRAN, AND J. STILLMAN. Semi-unification. *Theoretical Computer Science*, volume 81(2), pages 169-188, April 1991.

[Katiyar & Sankar, 1992] D. KATIYAR AND S. SANKAR. Completely bounded quantification is decidable. In *ML Workshop*. ACM Press, June 1992.

[Kfoury & al., 1990] KFOURY, A., TIURYN, J., AND URZYCYZYN, P. The undecidability of the semi-unification problem. In proceedings of the *22th Symposium on Theory of Computing* (STOC). Baltimore, Maryland, May 1990.

[Kfoury & Tiuryn, 1990] KFOURY, A., AND TIURYN, J. Type reconstruction in finite rank fragments of the polymorphic lambda-calculus. In *the proceedings of the 1992's IEEE Conference on Logic in Computer Science*. IEEE Press, Oakland, June 1990.

[Krantz, 1987] KRANTZ, D. *ORBIT: An optimizing compiler for Scheme*. Ph. D. thesis. Yale University, New Haven, 1987.

[Larus & Hilfinger, 1988] LARUS, J. R., AND HILFINGER, P. N. Detecting conflicts between structure accesses. In *Proceedings of the 1988 ACM Conference on Programming Language Design and Implementation*. ACM, New-York, 1988.

[Leiss & Henglein, 1991] LEISS, H., AND HENGLEIN, F. A decidable case of semi-unification problem. In *Mathematical Foundations of Computer Science*, MFCS'1991, volume 520 of the *Lectures Notes in Computer Science*, pages 318-327. Springer-Verlag, 1991.

[Leroy, 1990] LEROY, X. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[Leroy & Weis, 1991] LEROY, X., AND WEIS, P. Polymorphic type inference and assignment. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1991.

[Leroy, 1992] LEROY, X. *Typage polymorphe d'un langage algorithmique*. Doctoral dissertation, Université Paris VII, 1992.

[Lincoln & Christian, 1989] LINCOLN, P., AND CHRISTIAN, J. Adventures in associative and commutative unification. In *Journal of Symbolic Computations*, volume 7, pages 217-240. Academic Press, 1989.

[Lucassen, 1987] LUCASSEN, J. M. Types and Effects, towards the integration of functional and imperative programming. *MIT/LCS/TR-408* (Ph. D. Thesis). MIT Laboratory for Computer Science, August 1987.

[Lucassen & Gifford, 1988] LUCASSEN, J. M., AND GIFFORD, D. K. Polymorphic Effect Systems. In *Proceedings of the 1988 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1988.

[Mac Cracken, 1979] MACCRACKEN, N. Investigation of a programming language with a polymorphic type structure. *Ph. D. Thesis*, Syracuse University, 1979.

[MacQueen, 1984] MACQUEEN, D. B. Modules for Standard ML. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 198-207. ACM Press, New-York, 1990.

[Mairson, 1990] MAIRSON, H. G. Deciding ML typability is complete in deterministic exponential time. In *Principles of Programming Languages 1989*. ACM Press, 1990.

[Michaylov & Pfenning, 1992] MICHAYLOV, S., AND PFENNING, F. Natural Semantics and some of its Meta-Theory in Elf. In *proceedings of the 2nd workshop on extensions of logic programming*. Lectures Notes in Computer Science, Springer Verlag, 1992.

[Milner, 1978] MILNER, R. A Theory for type polymorphism in programming. In *Journal of Computer and Systems Sciences*, Vol. 17, pages 348-375. 1978.

[Milner & al., 1990] MILNER, R., TOFTE, M., HARPER, R. The definition of Standard ML. *The MIT Press*, Cambridge, 1990.

[Milner, 1990] MILNER, R. *Communication and Concurrency*. Prentice-Hall, 1990.

[Milner, 1991] MILNER, R., AND TOFTE, M. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209-220, 1991.

[Milner & al., 1992] MILNER, R., PARROW, J., AND WALKER. D. A calculus of mobile processes. *Information and Computation*, 100(1):1-41, 1992.

[Mitchell & Harper, 1988] MITCHELL, J. C., AND HARPER, R. The Essence of ML. In *Proceedings of the 1988 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1988.

[Morris, 1968] MORRIS, J. H. Lambda Calculus Models of Programming Languages. *MAC-TR-57*. Massachusetts Institute of Technology, 1968.

[Naur, 1963] NAUR, P. (ED.) Revised report on the algorithmic language Algol 60. In *Communication of the ACM*, volume 1, pages 1-17. The ACM Press, 1963.

[Neirynck & al., 1989] NEIRYNCK, A., PANANGADEN, P., AND DEMERS, A. Effect analysis of higher-order languages. In *International Journal of Parallel Programming*, Vol. 18, No. 119. 1989.

[Odersky, 1992] ODERSKY, M. Observers for linear types. In *ESOP'92*, Lecture Notes In Computer Science, volume 582, pages 390-407. Springer Verlag, February 1992.

[O'Toole, 1989] O'TOOLE, J. W. Polymorphic type reconstruction. *Master Thesis*. MIT Laboratory for Computer Science, May 1989.

[O'Toole, 1990] O'TOOLE, J. W. Type abstraction rules for references: a comparison of four which have achieved notoriety. Technical Report 390, MIT Laboratory for Computer Science, 1990.

[Pierce, 1991] PIERCE, B. Bounded quantification is undecidable. In *Proc. 19th ACM Symp. Principles of Programming Languages*. ACM Press, 1992. *Technical Report CMU-CS-91-161*. Carnegie Mellon University, 1991.

[Plotkin, 1981] PLOTKIN, G. A structural approach to operational semantics. *Technical report DAIMI-FN-19*. Aarhus University, 1981.

[Randel, 1964] RANDELL, B., AND RUSSEL, L.J. *ALGOL 60 Implementation*. Academic Press, 1964.

[Rees & al., 1984] REES, J. A., ADAMS, N. I., AND MEEHAN, J. R. *The T Manual, fourth edition*. Yale University 1984.

[Robinson, 1965] ROBINSON, J. A. A machine oriented logic based on the resolution principle. In *Journal of the ACM*, Vol. 12(1), pages 23-41. ACM, New-York, 1965.

[Rosen, 1979] ROSEN, B. Data Flow Analysis for Procedural Languages. In *Journal of the ACM*, Vol. 26(2), pages 322-344. ACM, New-York, April 1979.

[Sabot, 1990] SABOT, G. W. *The paralation Model*. MIT Press 1990.

[Sabry & Felleisen, 1992] SABRY, A., AND FELLEISEN, M. Reasoning about programs in CPS. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 288-297. ACM Press, New-York, 1992.

[Samelson, 1960] SAMELSON, K., AND BAUER, F.L. Sequential formula translation. In *Communication of the ACM*, volume 3.2, pages 76-83. The ACM Press, February 1960.

[Rees & al., 1988] REES, J., AND CLINGER W., EDITORS. Fourth Report on the Algorithmic Language Scheme. September 1988.

[Rémy, 1991] RÉMY, D. Type inference for records in a natural extension of ML. Research report 1431, INRIA, 1991.

[Ruziska, 1991] RUZISKA, P. An efficient decision procedure for a class of set constraints. In *Mathematical Foundations of Computer Science*, MFCS'1991, volume 520 of the *Lectures Notes in Computer Science*, pages 415-425. Springer-Verlag, 1991.

[Siekmann, 1989] SIEKMANN, J. H. Unification theory. In *Journal of Symbolic Computations*, volume 7, pages 207-274. Academic Press, 1989.

[Sheldon & Gifford, 1990] SHELDON, A. M., AND GIFFORD, D. K. Static Dependent Types for First Class Modules. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming.* ACM, New-York, 1990.

[Steele, 1990] STEELE, G. L. *Common Lisp, the language.* Digital Press 1990.

[Stoy, 1977] STOY, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[Shivers, 1991] SHIVERS, O. Control-flow analysis of higher-order languages. *Ph. D. Thesis* and *Technical Report CMU-CS-91-145*, Carnegie Mellon University, Pittsburgh, May 1991.

[*Lisp, 1987] *Lisp Reference Manual, version 4.0. Thinking Machines Corporation, Cambridge, 1987.

[Stransky, 1988] STRANSKY, J. Analyse sémantique de structures de données dynamiques avec application au cas particulier de langages lispiens. *Doctoral Dissertation*, Université Paris-Sud, Orsay, June 1988.

[Talpin & Jouvelot, 1992] TALPIN, J. P., AND JOUVELOT, P. Polymorphic Type, Region and Effect Inference. In the *Journal of Functional Programming*, volume 2, number 3. Cambridge University Press, 1992.

[Talpin & Jouvelot, June 1992] TALPIN, J. P., AND JOUVELOT, P. The Type and Effect Discipline. In *the proceedings of the 1992's IEEE Conference on Logic in Computer Science.* Santa Cruz, California, June 1992.

[Talpin & Jouvelot, 1993] TALPIN, J. P., AND JOUVELOT, P. Compiling FX on the Connection Machine. *Research Report EMP-CRI-A208*, January 1993.

[Tang & Jouvelot, 1992] TANG, Y. M., AND JOUVELOT, P. Control-Flow Effects for Closure Analysis. In *proceedings of the 2nd Workshop on Semantics Analysis*, Bigre numbers 81-82, pages 313-321. Bordeaux, Octobre 1992.

[Tarsky, 1955] TARSKY, A. A lattice theoretical fixpoint theorem and its applications. In *Pacific Journal of Mathematics*, volume 5, pages 285-309. 1955.

[Tiuryn, 1990] TIURIN, J. Type inference problems, a survey. In *Mathematical Foundations of Computer Science*, MFCS'1990, volume 452 of the *Lectures Notes in Computer Science.* Springer-Verlag, 1991.

[Tofte, 1987] TOFTE, M. Operational semantics and polymorphic type inference. *PhD Thesis* and Technical Report *ECS-LFCS-88-54*, University of Edinburgh, 1987.

[Tofte, 1990] TOFTE, M. Type inference for polymorphic references. In *Information and Computation*, 89(1), pages 1-34, 1990.

[Tofte, 1992] M. TOFTE. Principal signatures for higher-order program modules. In *19th ACM Symposium on Principles of Programming Languages*, pages 189-199. ACM Press, 1992.

[Tofte & Talpin, 1993] TOFTE, M., AND TALPIN, J. P. A Theory of Stack Allocation in Polymorphically Typed Languages. Technical Report (*Draft*), University of Copenhagen, June 1993.

[Wadler, 1991] WADLER, P. Is there a mean for linear logic. In proceedings of the *ACM/IFIP symposium on partial evaluation and semantics based program manipulations*. Yale University, 1991.

[Wand, 1987] WAND, M. A simple algorithm and proof for type inference. In *Fundamenta Informaticae*, volume 10, pages 115-122. North Holland, 1987.

[Wand, 1991] WAND, M. On the correctness of procedure representation in higher-order languages. In *Mathematical Foundations of Computer Science*, MFCS'1991, volume 520 of the *Lectures Notes in Computer Science*. Springer-Verlag, 1991.

[Wand, 1992] WAND, M., AND OLIVA, D. P. Proving the correctness of storage representation. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 151-160. ACM, New-York, 1992.

[Wright & Felleisen, 1992] WRIGHT, A. K., AND FELLEISEN, M. A syntactic approach to type soundness. Technical Report *TR91-160*, Rice University, 1992.

[Wright, 1992] WRIGHT, A. K. Typing References by Effect Inference. In *the proceedings of the 1992's European Symposium on Programming*, volume 582 of the *Lectures Notes in Computer Science*, pages 473-491. Springer Verlag, 1992.

Appendix A

Typing Effects for Free

In the preceding chapters, we used constraint sets to relate types and effects in the reconstruction algorithm. In this chapter, we introduce a syntax of type and effect that is used both in the static semantics and in the algorithm.

In [Leroy, 1992], chapter 2, the author also presents a syntax for function types annotated with sets of informations. It is based on the notion of extension variables, used to type records [Rémy, 1991].

Our approach is more involved. We want to be as close as possible to a set algebra and, however, make as little change to it so as the problem of unifying terms in this algebra be unitary, as in a type system for extensible records.

Syntax

To achieve this goal, the syntax of effects implements an encoding of constraint sets into types. This encoding consist of distinguishing the latent effect of functions and the effect of expressions.

$\epsilon ::= \varsigma \mid \epsilon \cup \sigma$	latent effect
$\tau ::= \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid \alpha \mid \mathit{ref}_\rho(\tau) \mid \tau \xrightarrow{\epsilon} \tau$	types

Function Type and Latent Effect

The abstract syntax of type terms in the static semantics supports basic data types, type variables, reference types and function types. Latent effects, written ϵ, appear on the arrow of function types which consist of an extension variable ς (a kind of label) and an effect σ.

An Algebra for Latent Effects

In order to precisely identify the label ς associated with a function type, we imposes that the extension variable ς appears on the left of the form $\varsigma \cup \sigma$. This is achieved by restricting the axioms of the algebra of latent effects.

$(\epsilon \cup \sigma) \cup \sigma' = \epsilon \cup (\sigma \cup \sigma') = \epsilon \cup \sigma \cup \sigma'$	associativity
$\epsilon \cup \sigma \cup \sigma = \epsilon \cup \epsilon \cup \sigma = \epsilon \cup \sigma$	idempotence
$\epsilon \cup \sigma \cup \sigma' = \epsilon \cup \sigma' \cup \sigma$	right commutativity

Axioms for Latent Effects

Accordingly, substitutions θ will map effect variables to latent effects. The combination of substituted latent effect terms in effect terms is done in the obvious way.

Example Let $\theta = \{\varsigma \mapsto \epsilon'\}$ the substitution of the effect variable ς by the latent effect $\epsilon' = \varsigma' \cup \sigma'$. Thus, $\theta(\epsilon \cup \varsigma \cup \sigma) = (\epsilon \cup (\epsilon') \cup \sigma) = \epsilon \cup \varsigma' \cup \sigma' \cup \sigma$ and $\theta(\varsigma' \cup \sigma) = (\epsilon' \cup \sigma) = \varsigma' \cup \sigma' \cup \sigma$ ■

Static Semantics

In the static semantics, latent effects are constructed by the rule (abs), for lambda-expressions. They are propagated with other effects by the rule (app), which applies for application expressions.

$$\frac{\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau\} \vdash \mathtt{e} : \tau', \sigma}{\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \tau \xrightarrow{\epsilon \cup \sigma} \tau', \emptyset} \quad \text{(abs)} \qquad \frac{\mathcal{E} \vdash \mathtt{e} : \tau \xrightarrow{\epsilon} \tau', \sigma \quad \mathcal{E} \vdash \mathtt{e'} : \tau, \sigma'}{\mathcal{E} \vdash \mathtt{(e\ e')} : \tau', \epsilon \cup \sigma \cup \sigma'} \quad \text{(app)}$$

Syntax-Directed Static Semantics

Constraint-Less Reconstruction Algorithm

In this section, we present the constraint less inference algorithm $\mathcal{I}$. Given a type environment $\mathcal{E}$ and an expression $\mathtt{e}$, it returns the principal type τ and effect σ of $\mathtt{e}$ and a substitution θ which ranges over the free variables of $\mathcal{E}$.

$$\begin{array}{rl}
\mathcal{I}(\mathcal{E},\mathtt{e}) = & \text{case } \mathtt{e} \text{ of} \\
\mathtt{x} \Rightarrow & \text{if } \mathtt{x} \in Dom(\mathcal{E}) \text{ then } (Id, Inst(\mathcal{E}(\mathtt{x})), \emptyset) \text{ else fail} \\
\mathtt{(lambda\ (x)\ e)} \Rightarrow & \text{let } \alpha, \varsigma \text{ new and } (\theta, \tau, \sigma) = \mathcal{I}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}, \mathtt{e}) \\
& \text{in } (\theta, \theta\alpha \xrightarrow{\varsigma \cup \sigma} \tau, \emptyset) \\
\mathtt{(e\ e')} \Rightarrow & \text{let } (\theta, \tau, \sigma) = \mathcal{I}(\mathcal{E}, \mathtt{e}),\ (\theta', \tau', \sigma') = \mathcal{I}(\theta\mathcal{E}, \mathtt{e'}),\ \alpha, \varsigma \text{ new and } \theta'' = \mathcal{U}(\theta'\tau, \tau' \xrightarrow{\varsigma} \alpha) \\
& \text{in } (\theta'' \circ \theta' \circ \theta, \theta''\alpha, Observe(\theta''(\theta'(\theta\mathcal{E})), \theta''\alpha)(\theta''(\varsigma \cup \sigma' \cup \theta'\sigma))) \\
\mathtt{(let\ (x\ e)\ e')} \Rightarrow & \text{let } (\theta, \tau, \sigma) = \mathcal{I}(\mathcal{E}, \mathtt{e}) \text{ and } (\theta', \tau', \sigma') = \mathcal{I}(\theta\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto Gen(\theta\mathcal{E}, \sigma)(\tau)\}, \mathtt{e'}) \\
& \text{in } (\theta' \circ \theta, \tau', Observe(\theta'(\theta\mathcal{E}), \tau')(\theta'\sigma \cup \sigma'))
\end{array}$$

Reconstruction Algorithm $\mathcal{I}$

This new algorithm is strictly simpler than the one that we presented in chapter 3. This is mainly due to the absence of constraint sets.

$$\begin{array}{rcl}
Gen(\mathcal{E})(\tau) & = & \text{let } \vec{v} = fv(\tau) \setminus fv(\mathcal{E}) \text{ in } \forall\vec{v}.\tau \\
Inst(\forall\vec{v}.\tau) & = & \text{let } \vec{v'} \text{ fresh and } \theta = \{\vec{v} \mapsto \vec{v'}\} \text{ in } \theta\tau
\end{array}$$

Generalization and Instantiation

The algorithm $\mathcal{I}$ uses two functions *Gen* and *Inst* described above for generalizing and instantiating types. It uses the observation criterion defined in section 3.1.

Constraint-Less Unification Algorithm

The inference algorithm $\mathcal{I}$ uses the following unification procedure which is an extension of the algorithm $\mathcal{U}$ of chapter 2 unifying latent effects and description identifiers.

The algorithm $\mathcal{U}$ differs from the unification algorithm of chapter 2 by the unification of latent effects $(\varsigma \cup \sigma, \varsigma' \cup \sigma')$, which check that the effect equation supplied is not recursive, and then unifies the effect variables ς and ς' and sums up the related effects σ and σ'.

$$
\begin{array}{rcl}
\mathcal{U}(\tau,\tau') & = & \text{case } (\tau,\tau') \text{ of} \\
(\mathit{unit},\mathit{unit}) & \Rightarrow & \mathit{Id} \\
(\alpha,\alpha') & \Rightarrow & \{\alpha \mapsto \alpha'\} \\
(\alpha,\tau)|(\tau,\alpha) & \Rightarrow & \text{if } \alpha \in \mathit{fv}(\tau) \text{ then fail else } \{\alpha \mapsto \tau\} \\
(\mathit{ref}_{\varrho}(\tau),\mathit{ref}_{\varrho'}(\tau')) & \Rightarrow & \text{let } \theta = \{\varrho \mapsto \varrho'\} \text{ in } \mathcal{U}(\theta\tau,\theta\tau') \circ \theta \\
(\tau_i \stackrel{\varsigma\cup\sigma}{\rightarrow} \tau_f\,,\tau'_i \stackrel{\varsigma'\cup\sigma'}{\rightarrow} \tau'_f) & \Rightarrow & \text{if } \varsigma \in \mathit{fv}(\sigma') \text{ or } \varsigma' \in \mathit{fv}(\sigma) \text{ then fail else} \\
 & & \quad \text{let } \theta = \{\varsigma \mapsto \varsigma',\ \varsigma' \mapsto \varsigma' \cup \sigma \cup \sigma'\} \text{ and } \theta' = \mathcal{U}(\theta\tau_i\,,\theta\tau'_i) \circ \theta \\
 & & \quad \text{in } \mathcal{U}(\theta'\tau_f\,,\theta'\tau''_f) \circ \theta' \\
\text{otherwise} & \Rightarrow & \text{fail}
\end{array}
$$

Unification Algorithm

The correctness of algorithm $\mathcal{U}$ can be stated in the same terms as the theorem 2.8, because the unification problem of latent effect terms is unitary.

Proposition A.1 (Correctness of $\mathcal{U}$) *Let τ and τ' be two type terms in the domain of $\mathcal{U}$. If $\mathcal{U}(\tau,\tau') = \theta$, then $\theta\tau = \theta\tau'$ and, whenever $\theta'\tau = \theta'\tau'$, there exists a substitution θ'' such that $\theta' = \theta'' \circ \theta$.*

Proof By induction on type terms in the way of [Robinson, 1965]□

Correctness of the Reconstruction Algorithm

The main novelty of this algorithm is that it does not use constraints, because of to our representation for latent effects and substitutions. By getting rid of constraint sets, the statement and proof of the correctness theorems of the algorithm are significantly simplified.

Theorem A.1 (Soundness of $\mathcal{I}$) *If $\mathcal{I}(\mathcal{E},\mathtt{e}) = (\theta,\tau,\sigma)$ then $\theta\mathcal{E} \vdash \mathtt{e} : \tau,\sigma$.*

Case of (var) By hypothesis, $\mathcal{I}(\mathcal{E},\mathtt{x}) = (\mathit{Id},\tau,\emptyset)$. By definition of $\mathcal{I}$, $\tau = \mathit{Inst}(\mathcal{E}(\mathtt{x}))$ and thus $\tau \preceq \mathcal{E}(\mathtt{x})$. By definition of (var), $\mathcal{E} \vdash \mathtt{x} : \tau,\emptyset$.

Case of (abs) By hypothesis, $\mathcal{I}(\mathcal{E},\mathtt{(lambda\ (x)\ e)}) = (\theta,\theta\alpha \stackrel{\varsigma\cup\sigma}{\rightarrow} \tau,\emptyset)$. By definition of $\mathcal{I}$, α and ς are fresh and $\mathcal{I}(\mathcal{E}_{\mathtt{x}}+\{\mathtt{x} \mapsto \alpha\},\mathtt{e}) = \mathcal{I}(\mathcal{E}_{\mathtt{x}}+\{\mathtt{x} \mapsto \alpha\},\mathtt{e})$. By induction hypothesis on $\mathtt{e}$, $\theta(\mathcal{E}_{\mathtt{x}}+\{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \tau,\sigma$. By definition of (abs), $\theta\mathcal{E} \vdash \mathtt{(lambda\ (x)\ e)} : \theta\alpha \stackrel{\varsigma\cup\sigma}{\rightarrow} \tau,\emptyset$.

Case of (app) By hypothesis, $\mathcal{I}(\mathcal{E},\mathtt{(e\ e')}) = (\theta'' \circ \theta' \circ \theta,\theta''\alpha,\mathit{Observe}(\theta''(\theta'(\theta\mathcal{E})),\theta''\alpha)(\theta''(\varsigma \cup \sigma' \cup \theta'\sigma)))$. By definition of $\mathcal{I}$,

$$(\theta,\tau,\sigma) = \mathcal{I}(\mathcal{E},\mathtt{e}) \qquad (\theta',\tau',\sigma') = \mathcal{I}(\theta\mathcal{E},\mathtt{e'}) \quad \text{and} \quad \theta'' = \mathcal{U}(\theta'\tau,\tau' \stackrel{\varsigma}{\rightarrow} \alpha)$$

where α and ς are fresh. By induction hypothesis on $\mathtt{e}$, $\theta\mathcal{E} \vdash \mathtt{e} : \tau,\sigma$. By the lemma 3.4,

$$\theta''(\theta'(\theta\mathcal{E})) \vdash \mathtt{e} : \theta''(\theta'\tau),\theta''(\theta'\sigma)$$

By induction hypothesis on $\mathtt{e'}$, $\theta'(\theta\mathcal{E}) \vdash \mathtt{e'} : \tau',\sigma'$. By the lemma 3.4,

$$\theta''(\theta'(\theta\mathcal{E})) \vdash \mathtt{e} : \theta''\tau',\theta''\sigma'$$

By the lemma A.1, $\theta''(\theta'\tau) = \theta''(\tau' \stackrel{\varsigma\cup\emptyset}{\rightarrow} \alpha)$ and by definition of the rule (app),

$$\theta''(\theta'(\theta\mathcal{E})) \vdash \mathtt{(e\ e')} : \theta''\alpha,\mathit{Observe}(\theta''(\theta'(\theta\mathcal{E})),\theta''\alpha)(\theta''(\theta'\sigma \cup \sigma' \cup \varsigma))$$

Case of (let) By hypothesis, $\mathcal{I}(\mathcal{E}, \texttt{(let (x e) e')}) = (\theta' \circ \theta, \tau', \mathit{Observe}(\theta'(\theta\mathcal{E}), \tau')(\sigma' \cup \theta'\sigma))$. By definition of $\mathcal{I}$,

$$(\theta, \tau, \sigma) = \mathcal{I}(\mathcal{E}, \texttt{e}) \quad \text{and} \quad (\theta', \tau', \sigma') = \mathcal{I}(\theta\mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \mathit{Gen}(\theta\mathcal{E}, \sigma)(\tau)\}, \texttt{e'})$$

By induction hypothesis on $\texttt{e}$, $\theta\mathcal{E} \vdash \texttt{e} : \tau, \sigma$. By the lemma 3.4,

$$\theta''(\theta'(\theta\mathcal{E})) \vdash \texttt{e} : \theta''(\theta'\tau), \theta''(\theta'\sigma)$$

By induction hypothesis on $\texttt{e'}$ with $\theta\mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \mathit{Gen}(\theta\mathcal{E}, \sigma)(\tau)\}$, $\theta'(\theta\mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \mathit{Gen}(\theta\mathcal{E}, \sigma)(\tau)\}) \vdash \texttt{e'} : \tau', \sigma'$. By definition of the rule (let),

$$\theta'(\theta\mathcal{E}) \vdash \texttt{(let (x e)e')} : \tau', \theta'\sigma \cup \sigma' \ \square$$

Theorem A.2 (Completeness of $\mathcal{I}$) *If $\theta''\mathcal{E} \vdash e : \tau', \sigma'$ then $\mathcal{I}(\mathcal{E}, \texttt{e}) = (\theta, \tau, \sigma)$ and there exists θ' such that $\theta''\mathcal{E} = \theta'\theta\mathcal{E}$, $\tau' = \theta'\tau$ and $\sigma' = \theta'\sigma$.*

Proof The proof is by induction on the structure of expressions.

Case of (var) By hypothesis $\theta''\mathcal{E} \vdash \texttt{x} : \tau, \sigma$. By definition of (var), $\mathcal{E}(\texttt{x}) = \forall\vec{v}.\tau''$ and $\tau \preceq \theta''\mathcal{E}(\texttt{x})$. By definition of $\preceq$, there exists θ_1 defined on $\vec{v}$ such that $\tau = \theta_1\tau''$. By definition of the algorithm $\mathcal{I}$, $\tau' = \mathit{Inst}(\mathcal{E}(\texttt{x}))$ and $\mathcal{I}(\mathcal{E}, \texttt{x}) = (\mathit{Id}, \tau', \emptyset)$. By definition of *Inst*, $\theta = \{\vec{v} \mapsto \vec{v'}\}$, the $\vec{v'}$ are fresh and $\tau' = \theta'_1\tau''$. Thus, $\theta' = \theta''_{\vec{v'}} + \{\vec{v'} \mapsto \theta_1(\vec{v})\}$ satisfies $\tau = \theta'\tau'$.

Case of (let) By hypothesis, $\theta''\mathcal{E} \vdash \texttt{(let (x e}_1\texttt{) e}_2\texttt{)} : \tau', \sigma'_1 \cup \sigma'_2$. By definition of (let),

$$\theta''\mathcal{E} \vdash \texttt{e}_1 : \tau'_1, \sigma'_1 \quad \text{and} \quad \theta''\overline{\mathcal{E}_{\texttt{x}}} + \{\texttt{x} \mapsto \mathit{Gen}(\sigma'_1, \theta''\mathcal{E})(\tau'_1)\} \vdash \texttt{e}_2 : \tau', \sigma'_2$$

By induction hypothesis on $\texttt{e}_1$, $(\theta_1, \tau_1, \sigma_1, \kappa_1) = \mathcal{I}(\mathcal{E}, \kappa, \texttt{e}_1)$ and there exists a substitution θ'_1 such that

$$\theta''\mathcal{E} = \theta'_1(\theta_1\mathcal{E}) \qquad \tau'_1 = \theta'_1\tau_1 \quad \text{and} \quad \sigma'_1 \supseteq \theta'_1\sigma_1$$

Since $\theta''\mathcal{E} = \theta'_1(\theta_1\mathcal{E})$, $\tau'_1 = \theta'_1\tau_1$, $\sigma'_1 \supseteq \theta'_1\sigma_1$ and by definition of *Gen*, $\tau \prec \mathit{Gen}(\sigma'_1, \theta''\mathcal{E})(\tau'_1)$ implies that $\tau \prec \mathit{Gen}(\theta'_1\sigma_1, \theta'_1(\theta_1\mathcal{E}))(\theta'_1\tau_1)$ for any τ. By the lemma 3.5,

$$\theta'_1(\theta_1\overline{\mathcal{E}_{\texttt{x}}}) + \{\texttt{x} \mapsto \mathit{Gen}(\theta'_1\sigma_1, \theta'_1(\theta_1\mathcal{E}))(\theta'_1\tau_1)\} \vdash \texttt{e}_2 : \tau', \sigma'_2$$

Let $\forall\vec{v}.\tau_1 = \mathit{Gen}(\sigma_1, \theta_1\mathcal{E})(\tau_1)$ and define θ''_1 by *Id* on $\vec{v}$ and by θ'_1 elsewhere. By definition, $\theta''_1(\theta_1\mathcal{E}) = \theta'_1(\theta_1\mathcal{E})$ and $\theta''_1\sigma_1 = \theta'_1\sigma_1$. Thus,

$$\theta''_1(\theta_1\mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \forall\vec{v}.\tau_1\}) \vdash \texttt{e}_2 : \tau', \sigma'_2$$

By induction hypothesis on $\texttt{e}_2$ with $\theta_1\mathcal{E}_{\texttt{x}}$, $\mathcal{I}(\theta_1\mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \forall\vec{v}.\tau_1\}, \texttt{e}_2) = (\theta_2, \tau, \sigma_2)$ and there exists a substitution θ'_2 such that $\tau' = \theta'_2\tau$, $\sigma'_2 \supseteq \theta'_2\sigma_2$ and

$$\theta''_1\overline{\theta_1\mathcal{E}_{\texttt{x}}} + \{\texttt{x} \mapsto \overline{\forall\vec{v}.(\tau_1, \kappa'_1)}\} = \theta'_2(\theta_2\overline{\theta_1\mathcal{E}_{\texttt{x}}} + \{\texttt{x} \mapsto \overline{\forall\vec{v}.(\tau_1, \kappa'_1)}\}) = \theta'_2(\theta_2\theta_1\mathcal{E}_{\texttt{x}} + \{\texttt{x} \mapsto \forall\vec{v}.(\overline{\kappa}'_1\tau_1)\})$$

Let us write $\theta = \theta_2 \circ \theta_1$, $\sigma_3 = \theta_2\sigma_1 \cup \sigma_2$ and $\sigma = \mathit{Observe}(\overline{\kappa}'(\theta\mathcal{E}), \overline{\kappa}'\tau)(\overline{\kappa}'\sigma_3)$. By definition of the algorithm, we get that:

$$\mathcal{I}(\mathcal{E}, \kappa, \texttt{(let (x e}_1\texttt{) e}_2\texttt{)}) = (\theta, \tau, \sigma, \kappa')$$

Let V be the set of free variables in $\theta\mathcal{E}$, τ_2 and σ_2. Define θ' by θ'_2 on V and θ''_1 otherwise. Thus,

$$\theta''\mathcal{E} = \theta'(\theta\mathcal{E}) \qquad \tau' = \theta'\tau \quad \text{and} \quad \sigma' \supseteq \theta'\sigma_3$$

Since $\theta'\sigma = \theta'\mathit{Observe}(\theta\mathcal{E}, \tau)(\sigma_3)$ and by the lemma 3.3, $\theta'\sigma \subseteq \mathit{Observe}(\theta'(\theta\mathcal{E}), \theta'\tau)(\theta'\sigma_3)$ and thus $\sigma' \supseteq \theta'\sigma$.

Case of (abs) By hypothesis, $\theta''\mathcal{E} \vdash \texttt{(lambda (x) e)} : \tau_i \xrightarrow{\epsilon' \cup \sigma'} \tau_f, \emptyset$. By definition of (abs), $\theta''\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \tau_i\} \vdash \mathtt{e} : \tau_f, \sigma'$. With a fresh variable α, this is equivalent to:

$$\theta'' \circ \{\alpha \mapsto \tau_i\}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}) \vdash \mathtt{e} : \tau_f, \sigma'$$

By induction hypothesis on $\mathtt{e}$, $\mathcal{I}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}, \mathtt{e}) = (\theta, \tau, \sigma)$ and there exists a substitution θ'_1 such that:

$$\theta'' \circ \{\alpha \mapsto \tau_i\}(\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}) = \theta'_1(\theta\mathcal{E}_{\mathtt{x}} + \{\mathtt{x} \mapsto \alpha\}),\ \tau_f = \theta'_1\tau \quad \text{and} \quad \sigma' \supseteq \theta'_1\sigma$$

By definition of the algorithm,

$$\mathcal{I}(\mathcal{E}, \texttt{(lambda (x) e)}) = (\theta, \theta\alpha \xrightarrow{\varsigma \cup \sigma} \tau, \emptyset)$$

where ς is new. Let us consider $\theta' = \theta'_1 \circ \{\varsigma \mapsto \epsilon' \cup \sigma'\}$. We conclude that

$$\theta''\mathcal{E} = \theta'(\theta\mathcal{E}) \qquad \tau_i \xrightarrow{\epsilon' \cup \sigma'} \tau_f = \theta'(\theta\alpha \xrightarrow{\varsigma \cup \sigma} \tau) \quad \text{and} \quad \sigma'' \supseteq \emptyset$$

Case of (app) The hypothesis is $\theta''\mathcal{E} \vdash \texttt{(e}_1\ \texttt{e}_2\texttt{)} : \tau', \sigma'_1 \cup \sigma'_2 \cup \sigma'_3$. By the definition of (app),

$$\theta''\mathcal{E} \vdash \mathtt{e}_1 : \tau'_2 \xrightarrow{\sigma'_3} \tau', \sigma'_1 \quad \text{and} \quad \theta''\mathcal{E} \vdash \mathtt{e}_2 : \tau'_2, \sigma'_2$$

By induction hypothesis on $\mathtt{e}_1$, we have $\mathcal{I}(\mathcal{E}\mathtt{e}_1) = (\theta_1, \tau_1, \sigma_1)$ and there exists θ'_1 such that

$$\theta''\mathcal{E} = \theta'_1(\theta_1\mathcal{E}) \qquad \tau'_1 = \tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_1\tau_1 \quad \text{and} \quad \sigma'_1 \supseteq \theta'_1\sigma_1$$

By induction hypothesis on $\mathtt{e}_2$, $\mathcal{I}(\theta_1\mathcal{E}, \mathtt{e}_2) = (\theta_2, \tau_2, \sigma_2)$ and there exists θ'_2 such that

$$\theta''\mathcal{E} = \theta'_2(\theta_2(\theta_1\mathcal{E})) \qquad \tau'_2 = \theta'_2\tau_2 \quad \text{and} \quad \sigma'_2 \supseteq \theta'_2\sigma_2$$

Let V be the set of free variables in $\theta_2(\theta_1\mathcal{E})$, τ_2 and σ_2. Take α and ς new and define θ'_3 by:

$$\theta'_3\upsilon = \begin{cases} \theta'_2\upsilon, & \upsilon \in V \\ \tau', & \upsilon = \alpha \\ \sigma'_3, & \upsilon = \varsigma \\ \theta'_1\upsilon, & \text{otherwise} \end{cases}$$

By definition, $\theta''\mathcal{E} = \theta'_3(\theta_2(\theta_1\mathcal{E}))$, $\tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_3(\tau_2 \xrightarrow{\varsigma} \alpha)$ and $\theta'_2\sigma_2 = \theta'_3\sigma_2$. Now, for every υ in τ_1 or σ_1, either υ is in $fv(\theta_1\mathcal{E})$ or υ is fresh. Thus, for every such υ in $fv(\theta_1\mathcal{E})$, since $\theta'_3(\theta_2(\theta_1\mathcal{E})) = \theta'_2(\theta_2(\theta_1\mathcal{E})) = \theta'_1(\theta_1\mathcal{E})$, we have $\theta'_3(\theta_2\upsilon) = \theta'_2(\theta_2\upsilon) = \theta'_1\upsilon$. Otherwise, υ is fresh and thus $\theta'_3(\theta_2\upsilon) = \theta'_3\upsilon = \theta'_1\upsilon$. Thus,

$$\tau'_2 \xrightarrow{\sigma'_3} \tau' = \theta'_3(\theta_2\tau_1) \quad \text{and} \quad \theta'_1\sigma_1 = \theta'_3(\theta_2\sigma_1)$$

Since $\theta'_3(\theta_2\tau_1) = \theta'_3(\tau_2 \xrightarrow{\varsigma} \alpha)$, by the lemma A.1, there exists a substitution θ_3 such that,

$$\theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha) \quad \text{and} \quad \theta_3(\theta_2\tau_1) = \theta_3(\tau_2 \xrightarrow{\varsigma} \alpha)$$

By the lemma A.1, there exists a substitution θ' satisfying $\theta_3\kappa_2$ such that $\theta'_3 = \theta' \circ \theta_3$. We conclude that

$$\theta''\mathcal{E} = \theta'(\theta\mathcal{E}) \qquad \tau'_1 = \theta'(\theta_3(\tau_2 \xrightarrow{\varsigma} \alpha)) \quad \text{and} \quad \sigma'_1 \cup \sigma'_2 \cup \sigma'_3 \supseteq \theta'(\theta_3\sigma_3)$$

By definition of $\mathcal{I}$,

$$\mathcal{I}(\mathcal{E}, \texttt{(e}_1\ \texttt{e}_2\texttt{)}) = (\theta_3 \circ \theta_2 \circ \theta_1, \theta_3\alpha, \mathit{Observe}(\overline{\kappa}'\theta\mathcal{E}, \overline{\kappa}'\tau)(\overline{\kappa}'\theta_3(\theta_2\sigma_1 \cup \sigma_2 \cup \varsigma)))$$

By the lemma 3.3, we conclude that $\sigma' \supseteq \theta' \mathit{Observe}(\overline{\kappa}'\theta\mathcal{E}, \overline{\kappa}'\tau)(\overline{\kappa}'\sigma_3)$ □

Résumé

Dans de nombreuses disciplines scientifiques, comme en informatique, et tout spécialement dans le domaine des langages de programmation, il est communément acquis d'associer aux études théoriques, sur les modèles de calcul, des travaux de nature plus pratique de mise en oeuvre, ou d'implémentation. Pour cela, les techniques de mise en œuvre des langages de programmation reposent sur des méthodes formelles qui permettent d'en assurer la correction.

Parmi ces méthodes formelles, le *typage statique* est sans doute la forme la plus populaire. Il consiste à détecter, au moment de la compilation, une source fréquente d'erreurs d'exécution de programmes: l'usage inconsistant d'une valeur par rapport à la structure de cette valeur. Dès lors que le typage statique d'un programme est vérifié, aucune erreur d'accès aux données ne peut se produire pendant son exécution.

Dans la recherche d'un compromis entre simplicité et performance, l'introduction du typage *polymorphe* est à l'origine de progrès notables, permettant le typage statique des fonctions génériques. Le typage polymorphe est parfaitement approprié pour les langages fonctionnels. Mais l'ajout de traits de programmation impérative à un langage fonctionnel s'accompagne de la nécessité d'introduire une notion d'*état* pour comprendre le sens des programmes.

Les *systèmes d'effet* permettent d'intègrer le typage polymorphe et la programmation impérative. De même qu'un type représente ce qu'un programme calcule, un effet décrit comment ce programme calcule. Types et effets sont annotés par des *régions*. Les régions décrivent des relations de partage entre les zones mémoire où résident les structures de données.

L'intérêt porté aux langages de programmation fonctionnels tels que ML ne se limite pas au seul problème du typage polymorphe, et notre système d'effet procure des informations utiles aussi bien pour le programmeur, qui peut par ce moyen décrire la spécification de ses applications, que pour le compilateur qui peut utiliser les informations de type pour produire un code plus efficace et une représentation de données moins coûteuse.

Abstract

In many scientific disciplines as well as in computer science, but perhaps more specifically in the area of programming languages, it is a widely established fact that theoretical research must be connected and validated with practical investigations and implementation techniques. To achieve this goal, it is very important to base the development of implementation techniques for programming languages on methods that allow them to be formalized simply and prove them correct.

Among program analysis methods, static typing is the most popular technique. It detects a common cause of execution errors in a program: the inconsistent use of a data structure. Its strength is that the successful type checking of a program guarantees the absence of type errors.

In the quest for simplicity and expressiveness, the introduction of type polymorphism resulted in a notable progress by allowing the static typing of generic functions. Polymorphic typing is appropriate for functional programming languages. However, adding imperative features to a functional language necessitates to introduce a notion of state to understand the meaning of programs.

Effect systems permit to integrate imperative constructs to polymorphic functional languages. Just as types describe the structure of what expressions compute, effects describe how expressions compute. Types and effects are decorated with regions. A region describe an uniform sharing relation between data structures and thus helps to figure out how storage resources are used and distributed in a program.

The academic interest in functional programming does not limit itself to the sole topic of polymorphic type inference. Polymorphic effect system provides useful information for both the programmer, who can describe the intended specification of its programs, and the compiler, which can use types to generate more efficient code and represent data better.