No d'ordre

THESE de DOCTORAT de L'UNIVERSITE PARIS VI

Spécialité

Systèmes Informatiques

Présentée

par Yan-Mei TANG

pour obtenir le titre de DOCTORAT de L'UNIVERSITE PARIS VI

Sujet de la thèse :

Systèmes d'Effet et Interprétation Abstraite pour l'Analyse de Flot de Contrôle

Soutenue le 28 Mars 1994

devant le jury composé de :

MM.	Claude	Girault	Président
	Charles	Consel	Rapporteur
	Michel	Mauny	Rapporteur
	Flemming	Nielson	Rapporteur
	Paul	Feautrier	Examinateur
	Pierre	Jouvelot	Examinateur

ECOLE NATIONALE SUPERIEURE DES MINES DE PARIS

Rapport A/258/CRI

Résumé

L'analyse du flot de contrôle est une technique d'importance majeure pour la compilation efficace des langages de programmation fonctionnelle. Cette technique consiste, au moment de compiler un programme, à calculer une approximation de son graphe d'appel, c'est à dire quelles fonctions sont utilisées par le programme, dans quel ordre et à quel niveau. Bien entendu, la précision de telles informations subordonne le possible gain en performance d'exécution résultant de leur l'utilisation au moyen d'un compilateur optimisant. Dans cette thèse, nous considérons deux approches théoriques pour formaliser l'analyse de flot de contrôle: *l'inférence d'effet* et *l'interprétation abstraite*. Nous étendons ces deux méthodes d'analyse statique pour ensuite montrer comment les combiner efficacement dans le cadre de l'analyse de flot de contrôle.

La première méthode que nous présentons, l'inférence d'effet, peut être définie comme une extension des techniques qui sont utilisées pour la vérification du typage des programmes en ML. Ainsi, de la même manière qu'un type représente l'approximation d'une valeur, un *effet* représente l'approximation d'une action ou d'une transition d'état. Ici, plutôt que de nous limiter à adapter les techniques de typage *à la ML* pour faire de l'analyse de flot de contrôle, nous introduisons une notion avancée de *sous-typage* qui permet d'augmenter notablement la flexibilité et la précision de cette technique.

La seconde méthode que nous considérons est l'interprétation abstraite qui, utilisant des techniques d'approximation de point-fixe, permet notamment d'obtenir des informations plus précises (sur les fonctions récursives, en particulier), là où d'autres techniques utilisent des méthodes de calcul beaucoup plus simples. Nous proposons de combiner ces deux techniques, introduisant la notion *d'analyse sémantique séparée*. Nous proposons d'associer une analyse de programme basée sur l'interprétation abstraite, profitant ainsi d'une technique performante pour les expressions closes, avec l'utilisation d'un système d'effet, afin de pouvoir spécifier des informations statiques en présence de compilation séparée, et donc d'information partielle sur l'ensemble d'un programme.

Enfin, nous étudions l'application de l'analyse de flot de contrôle pour la *détection d'échappements*, c'est à dire la détection, en ML, des variables dont l'existence dépasse le cadre lexical de leur définition. Détecter cette information permet au compilateur de choisir une stratégie optimale pour l'allocation des fonctions. Ainsi, lorsqu'une variable n'est jamais référencée en dehors du cadre lexical de sa définition, on peux alors la représenter dans la pile d'exécution. Dans le cas contraire, elle doit être représentée dans le *tas*, c'est à dire gérée par le sous-système de gestion mémoire.

Abstract

Control-flow analysis is important in optimizing compilers of functional languages. It strives to approximate at compile time dynamic function call graphs of program evaluation. The precision of the approximation drives the efficiency of optimizations applied in compilers. Control-flow analysis can be expressed by both *effect systems* and *abstract interpretation*. My thesis work extends and combines these two static analysis approaches and describes their applications to control-flow analysis.

Effect systems extend classical ML type systems with effect information. Just like types describe the possible values of expressions, effects describe their evaluation behaviors. My thesis introduces *subtyping* to improve both flexibility and accuracy of effect systems. The subtype relation is limited to the inclusion relation on effects. Control-flow analysis by effect systems collapses different call contexts together, thus limiting the accuracy of control-flow information.

Abstract interpretation is built upon denotational semantics by approximating the fixpoint nature of the language semantics. If abstract interpretation performs more precise static analysis due to its more operational nature, effect systems support separate compilation more naturely thanks to module signatures. My thesis introduces the new notion of *separate abstract interpretation* that extends abstract interpretation in the context of separate compilation based on the type and effect information of module signatures. It makes the control-flow analysis as effective as the abstract interpretation approach on closed expressions, but is also able to tackle expressions with free variables by using their types to approximate their abstract values.

Finally, my thesis studies the application of control-flow analysis to *escape analysis*. This analysis identifies the free variables that outlive the lexical scope of function definitions, thus helping compilers choose an efficient closure allocation strategy. Non-escaping variables can be safely allocated in the stack, while heap-allocation is only used for escaping ones.

Remerciements

Je remercie Claude Girault, Professeur à l'Université de Paris 6, d'avoir voulu présider mon jury.

Je remercie Charles Consel, Professeur à l'Université de Rennes, Michel Mauny, Chargé de Recherche à l'INRIA et Flemming Nielson, Professeur à l'Université de Aarhus pour m'avoir fait l'honneur d'accepter d'être mes rapporteurs.

Je remercie Paul Feautrier, Professeur à l'Université de Versailles, pour s'être montré un directeur de thèse agréable et arrangeant.

Je remercie Pierre Jouvelot, Chargé de Recherche à l'Ecole des Mines de Paris, qui m'a accueillie au CRI (Centre de Recherche en Informatique) il y a trois ans et encadré mon travail en y contribuant très largement. Grand merci à lui, car encore, sans lui cette thèse ne serait pas.

Je remercie Michel Lenci, ancien directeur de CRI, de qui je garde le souvenir d'un directeur unique par sa gentillesse et sa sympathie.

Je remercie Jacqueline Altimira, Corinne Ancourt, Vincent Dornic, Francois Irigoin, Marie-Thérèse Lesage, Francois Masdupuy, Nadine Oliver et tous mes autres collègues du CRI pour leur aide et amitié.

Je remercie W. Kluge, Professeur à l'Université de Kiel, qui m'a donné une chance unique de pouvoir venir en Europe il y a cinq ans.

Je remercie mes amis, Hai-fu, Zhen-wu Zhang et Lian Gu, qui m'ont beaucoup aidée et soutenue.

Enfin, je dédie cette thèse à mes parents en Chine, Gui-Zhen et De-Xin, mes beaux-parents en France, Annie et Maurice, à mon mari Jean-Pierre et à notre fille Julie.

Contents

Chapter 1

Introduction

L'analyse de flot de contrôle joue un rôle-clé dans les compilateurs optimisants pour les langages de programmation fonctionnelle.

Motivation

Les langages de programmation fonctionnelle, comme Lisp [Steele90], Scheme [Rees88] et ML [Mitchell88, Appel90, Milner90] sont largement reconnus pour leur caractère expressif et sémantique simple et bien-fondée. Cependant, il est beaucoup plus difficile de compiler efficacement ces langages, à l'instar d'autres langages plus traditionnels, tels que Fortran ou C. De fait, il existe des différences importantes quant au niveau d'optimisation que tel compilateur est capable d'opérer sur un programme, considérant distinctement ces deux classes de langages de programmation. En particulier, un compilateur pour un langage impératif traditionnel emploie typiquement une variété de techniques d'optimisation inter-procédurales du flot de données, telles que la propagation de constantes, l'élimination des variables non-utilisées, l'élimination des variables dites d'induction, etc [Aho86].

Toutes ces optimisations dépendent, bien entendu, de la bonne connaissance du graphe de flot de contrôle au moment de la compilation. Malheureusement, dans les langages de programmation fonctionnelle, comme les fonctions sont des valeurs à part entière, c'est à dire pouvant être passées en paramètre ou en résultat à par d'autres fonctions, le graphe de flot de contrôle ne peut pas être déterminé avec exactitude au moment de la compilation.

L'imprécision avec laquelle le graphe de flot de contrôle peut être déterminé dans les langages de programmation fonctionnelle rend bien entendu plus difficile, au moment de compiler un programme, les optimisations basées sur une connaissance globale du flot de données. Cette optimisation est pourtant d'importance car, pour ces langages, une fonction est représentée par son code ainsi qu'un environnement, c'est à dire une structure de données qui contient les valeurs de ses variables libres, *capturées* au moment de sa création. C'est pourquoi de nombreux travaux ont déjà porté sur la détermination du graphe de flot de contrôle afin d'optimiser leur allocation ou leur représentation [Leroy90-1, Appel89, Appel92, Tang92, Wand93]. L'analyse du flot de contrôle permet donc d'effectuer de telles optimisations. Elle peut être exprimée à l'aide de l'interprétation abstraite [Shivers91] ou bien à l'aide de l'inférence d'effet [Tang92].

Les systèmes d'effet utilisent un système de règles comme outil de spécification d'analyse statique. Utiliser un système de règles permet d'établir un jugement des programmes au regard d'un certain critère. Ce moyen d'expression est facile à comprendre et permet donc de raisonner sur les propriétés des programmes. L'utilisation d'un système de règles comme outil de spécification permet également d'isoler plus précisément ce qui est du ressort de l'implémentation : l'algorithme. La distinction entre spécification et réalisation est notamment avantageuse, s'agissant d'étendre l'analyse statique à de nouveaux traits du langage. A l'instar de cette méthode, l'interprétation abstraite ne permet pas une si précise distinction entre spécification et réalisation. Cela rend beaucoup plus difficile la formalisation et la réalisation d'un compilateur optimisant.

Un autre avantage d'un système d'effet par rapport à une interprétation abstraite est l'extension vers un langage de modules. Il est beaucoup plus naturel d'étendre un système d'effet à un langage de module donné (sur des principes similaires au typage) qu'une interprétation abstraite.

Cependant, il est clair que l'interprétation abstraite, utilisant des techniques d'approximation de point-fixe, permet d'obtenir des informations beaucoup plus précises (en particulier sur les fonctions récursives) là où d'autres techniques utilisent des méthodes de calcul beaucoup plus simples, et la précision de telles informations subordonne bien entendu le possible gain en performance d'exécution résultant de leur l'utilisation au moyen d'un compilateur optimisant.

Afin d'évaluer la relative performance de différents systèms de l'analyse de flot de contrôl, [Shivers91] propose de classifier la précision d'une analyse sémantique au moyen d'une notion d'ordre, qui indique en fait le nombre d'appels de fonctions successifs pris en compte dans l'analyse d'une expression donnée. Relativement à cet ordre, nous présentons deux types d'analyses, basées sur les systèmes d'effet et l'interprétation abstraite, qui sont respectivement d'ordre 0 et 1. Nous étudions ensuite l'application de ces techniques à l'analyse de flot de contrôle afin d'optimiser l'allocation des fonctions en ML.

Les Systèmes d'Effet

La vérification du typage dans les langages de programmation fonctionnelle est un domaine de recherche très actif et bien développé [Milner78, MacCracken79, Damas82, Cardelli85, Tofte90, O'Toole90]. L'utilisation de types afin de décrire la structure des valeurs manipulées dans un programme permet à l'utilisateur de comprendre, éventuellement de fournir explicitement, la spécification de son application, mais tend également à favoriser une programmation mieux structurée, et en tout cas à permettre la vérification des erreurs de typage ainsi qu'une meilleure représentation des valeurs manipulées dans le programme.

Nous disons qu'un langage de programmation a un typage statique fort lorsque le compilateur vérifie, contraint et détermine le typage des programmes. Cela permet notamment de prévenir, au moment de la compilation, de l'usage éventuellement incohérent des valeurs manipulées par le programme. En ce sens, le typage statique est la technique d'analyse statique la plus populaire. Cependant, l'utilisation du typage statique impose des contraintes assez fortes sur l'écriture des programmes et peut donc tendre à limiter la souplesse et le pouvoir d'expression d'un langage de programmation.

En ce sens, l'introduction de notions telles que le polymorphisme [Milner78, Kanellakis89, Leroy91] ou le sous-typage [Cardelli88, Aiken93, Fuh88, Stansifer88] a permis d'aller dans le sens de plus de souplesse et d'un caractère plus expressif pour les langages de programmation fortement typés, en autorisant notamment le typage de fonctions génériques ou bien encore en proposant des contraintes plus souples pour leur utilisation.

L'inférence d'effet [Lucassen87, Lucassen88], peut être définie comme une extension des techniques qui sont utilisées pour la vérification du typage des programmes en ML [Milner78, Tofte87]. Ainsi, de la même manière qu'un type représente l'approximation d'une valeur, un *effet* représente l'approximation d'une action ou d'une transition d'état. Les systèmes d'effet permettent d'élargir notablement le spectre d'utilisation de méthodes de calcul existantes pour la vérification ou l'inférence de type. Cependant, les effets sont souvent représentés dans des algèbres plus complexes que celles utilisées pour les types. A ce jour, les systèmes d'effet sont utilisés dans de nombreux domaines d'application tels que l'analyse d'effets de bord [Talpin92-2, Talpin92-2], l'analyse de complexité [Dornic91], ou l'analyse de liaison statique (binding-time analysis) [Consel93].

Le type d'une fonction est usuellement représenté par $t' \rightarrow t$, où t' est le type de son paramètre formel et t celui de son résultat. Dans un système d'effet, nous notons $t' \xrightarrow{\mathtt{F}} t$ le type d'une fonction, et nous décrivons par `F` la transition d'état consécutive à l'utilisation d'une fonction de ce type: `F` est *l'effet latent* de la fonction. Comme l'effet latent d'une fonction est propagé à l'aide de son type, il est possible de lier statiquement l'abstraction d'une fonction au regard de cette propriété depuis sa définition jusqu'à son utilisation dans le programme.

Cependant, l'introduction de nouvelles sortes, les effets, dans le système de type du langage de programmation peut introduire de nouvelles contraintes quant au caractère typable des programmes du langage. Nombre de systèmes d'effet existants [Talpin92-1, Dornic91, Tang92] utilisent la notion de *sub-effecting* pour augmenter la flexibilité du système sur ce point précis. Cette notion consiste à admettre un effet plus grand là où cela est nécessaire afin préserver le typage d'une expression, limitant cependant la précision de l'information calculée.

A la place de cette notion, nous proposons une notion généralisée de sous-typage [Tang93]. Cette notion se définit comme une extension de la relation d'inclusion entre les effets à une relation d'ordre entre les types. Ainsi, à l'instar du *sub-effecting*, une fonction peut admettre autant de super-types (au sens de cette relation d'ordre) que nécessaire pour préserver le typage d'une expression. L'apport de cette notion de sous-typage est certain quant à la précision avec laquelle il devient alors possible d'inférer l'effet d'une expression. En particulier, l'analyse de flot de contrôle peut être réalisée au moyen de cette outils, en présence de fonctions d'ordre supérieures, de constructions impératives et de compilation séparée. Cependant, relativement à la classification de [Shivers91], sa précision est d'ordre 0.

L'Interprétation Abstraite

L'interprétation abstraite [Cousot77, Cousot79, Mycroft81] est un cadre théorique très puissant pour formaliser l'analyse statique de programmes. Cet outil théorique s'appuie sur un modèle de sémantique formel, sémantique dénotationnelle ou bien sémantique opérationnelle, afin de décrire strictement comment s'effectue l'exécution d'un programme.

De cette sémantique, dite *standard*, du langage est ensuite dérivée une sémantique dite *exacte*, qui met en évidence les propriétés que l'on cherche à calculer. Cette sémantique exacte peut, par exemple à l'aide de la sémantique dénotationnelle, être décrite à l'aide d'équations de point-fixe sur des structures ordonnées. Ensuite, l'interprétation abstraite consiste à déterminer un point-fixe de ces équations sémantiques, telles qu'elles apparaissent dans un programme donné. Ces équations décrivent récursivement certaines propriétés du programme et sont définies, non pas sur une algèbre rudimentaire comme pour les systèmes d'effet, mais sur des structures de treillis pouvant être assez complexes. C'est au moyen de fonctions d'abstraction et de concrétisation que s'établie ensuite un lien cohérent entre la sémantique exacte et la sémantique abstraite, celle qui exprime les propriétés du programme effectivement calculables.

Le procédé est, on le voit, assez complexe. D'autre part, comme l'interprétation abstraite consiste essentiellement à déterminer un point-fixe des équations exprimées dans un programme, là où l'inférence d'effet utilise l'unification, l'interprétation abstraite autorise le calcul de propriétés beaucoup plus précises que les systèmes d'effet. Ceci étant, les systèmes d'effet donnent au contraire une information beaucoup plus facile à comprendre et à utiliser. En particulier, il est facile de spécifier le type et l'effet d'une fonction dans l'interface d'un module, et de supporter ainsi la compilation séparée de manière plus naturelle.

L'interprétation abstraite a été étudiée dans les langages traditionnels (impératifs) [Cousot77, Cousot79] mais aussi les langages de haut-niveau (fonctionnels) [Mycroft81, Deutsch90, Shivers91], tels que l'analyse du critère strict (*Strictness analysis*) [Mycroft81], l'analyse d'échappement (*Escape analysis*) [Goldberg90], ou l'analyse du flot de contrôle [Shivers91]. L'analyse de flot de contrôle basée sur l'interprétation abstraite peut en particulier distinguer les différents points d'appel d'une fonction d'une manière plus précise, permettant ainsi de réaliser une analyse d'ordre n, mais au détriment d'une compilation séparée aisée.

Si une interprétation abstraite permet effectivement de calculer des informations très précises sur les programmes, l'utilisation d'un système de type et d'effet permet au programmeur de spécifier une approximation raisonnable des propriétés statiques de chacun des modules de son programme, interfacés au reste de l'application au moyen d'une signature [Sheldon90]. Ainsi, les systèmes d'effet sont des outils d'analyse plus souples et plus adaptés à la compilation séparée.

Afin de profiter de chacun de ces avantages, nous présentons dans cette thése une notion combinée d'*interprétation abstraite séparée* [Tang94], qui permet d'utiliser plus naturellement les techniques d'interprétation abstraite dans le contexte de la compilation séparée. Nous considérons le cadre de la compilation séparée comme la compilation des composants, ou modules, d'un programme en isolation les uns des autres. Nous formulons ainsi une analyse de flot de contrôle à l'aide de cette technique, qui permet d'obtenir des informations tout aussi précises que l'interprétation abstraite stan-

dard dans le cas d'un petit programme , mais qui est également capable de traiter efficacement les différents modules d'un programme plus important, en utilisant les information de type et d'effets de contrôle spécifiées dans l'interface des autres modules. Dans le cadre d'une analyse globale de flot de contrôle, l'utilisation d'un système de type et d'effet permet au programmeur de spécifier une approximation raisonnable des propriétés statiques de chacun des modules de son programme, interfacé au reste de l'application au moyen d'une signature. Cette signature utilise un système de type et d'effet afin de donner une approximation des informations de flot de contrôle pour le reste du programme.

L'Analyse d'Échappement

Enfin, nous présentons une application immédiate de l'analyse de flot de contrôle, qui consiste en l'optimisation de la représentation et de l'allocation des fonctions [Tang92]: l'analyse d'échappement. Cette technique consiste à identifier les variables d'une fonction pouvant être *capturées* par d'autres fonctions, et ainsi avoir une durée de vie, ou un cadre d'utilisation, plus grand que le cadre lexical de leur définition.

Cette information est d'importance pour la compilation et permet notamment de décider la classe de représentation des variables. Ainsi, lorsqu'une variable n'est jamais référencée en dehors du cadre lexical de sa définition, on peux alors la représenter dans la pile d'exécution. Dans le cas contraire, elle doit être représentée dans le *tas*, c'est à dire gérée par le sous-système de gestion mémoire.

Contributions

Dans cette thèse, nous présentons de nouvelles techniques permettant l'analyse de flot de contrôle dans les langages haut-niveau, sur principe de système d'effet et interprétation abstraite. Nous réalisons l'analyse de flot de contrôle basé sur les systèmes d'effet pour le compilateur Mini-FX.

Nous apportons une contribution notable au domaine des systèmes d'effet en introduisant une notion de sous-typage. Cette notion se définit comme une extension de la relation d'inclusion entre les effets à une relation d'ordre entre les types. Ainsi, à l'instar du *sub-effecting*, une fonction peut admettre autant de super-types (au sens de cette relation d'ordre) que nécessaire pour préserver le typage d'une expression. L'apport de cette notion de sous-typage est certain quant à la précision avec laquelle il devient alors possible d'inférer l'effet d'une expression. Nous montrons que l'introduction d'une règle de sous-typage dans un système d'effet permet de spécifier un problème de résolution décidable. Nous présentons un algorithme correct par rapport à cette spécification. Le cadre de ce travail se limite à un langage doté d'un système de type monomorphe. Il s'étend cependant de manière parfaitement naturelle en présence de types polymorphes, comme tout autre système de sous-typage doté des mêmes caractéristiques.

L'interprétation abstraite est une technique d'importante pour les analyses statiques des langages de programmation fonctinnelle. Si une interprétation abstraite permet effectivement de calculer des informations très précises sur les programmes, un système de type et d'effet peux etre untilisè dans le contexte de la compilation séparée. Afin de profiter de chacun de ces avantages, nous présentons la notion d'interprétation abstraite séparée qui permet de combiner les techniques d'analyse basées sur l'interprétation abstraite et l'inférence d'effet en un seul et même outil. L'utilisation d'un système d'effet permet de d'interfacer les différents modules d'un programme au moyen d'une spéfification du type et des effets de leurs variables exportées, permettant ainsi l'interprétation abstraite des différents modules du programme.

Nous étudions l'application de l'analyse de flot de contrôle à la détection des variables dont l'existence dépasse le cadre lexical de leur définition. Détecter cette information permet au compilateur de choisir une stratégie optimisée pour l'allocation des fonctions. Ainsi, lorsqu'une variable n'est jamais référencée en dehors du cadre lexical de sa définition, on peut alors la représenter dans la pile d'exécution. Nous définissons une machine abstraite dotée d'une pile de contrôle pour son exécution, et nous montrons comment les variables peuvent être représentées en pile lorsque les fonctions ne les capturent pas.

Résumé des Chapitres

Chapitre 3 : Un Système d'Analyse de Flot de Contrôle

L'analyse du flot de contrôle est une technique utilisée pour la compilation efficace des langages de programmation fonctionnelle. Elle consiste à calculer une approximation de son graphe d'appel de fonction du programme. Le flot de contrôle d'une expression peut par exemple être statiquement représenté par un ensemble de fonctions susceptibles d'être appelées durant l'exécution de l'expression. Nous présentons dans ce chapitre une première spécification de l'analyse de flot de contrôle utilisant un système d'effet.

Ce chapitre a pour but essentiel de servir d'introduction à un certain nombre de concepts qui seront plus amplement développés et améliorés dans les chapitres suivants. Nous définissons le cadre formel permettant de prouver notre spécification, au moins d'un critère de cohérence vis à vis de la sémantique dynamique du langage. Nous montrons que notre formulation de l'analyse de flot de contrôle à l'aide d'un système d'effet s'adapte très simplement à des traits variés des langages de haut-niveau: fonctions d'ordre supérieur, opérations de style impératif, compilation séparée. Cependant, ce premier système n'offre pas de performances remarquables en terme de précision ou de flexibilité, et nous tenterons dans les chapitres suivant d'améliorer cet outil en ce sens.

Chapitre 4 : Un Système d'Effet Flexible

Afin d'améliorer la flexibilité du système présenté dans le précédent chapitre, nous introduisons dans notre spécification de l'analyse de flot de contrôle une règle autorisant l'utilisation de la relation d'inclusion sur les effets de contrôle. Ainsi, si une expression admet un effet $\mathtt{F}$, elle admet tout effet $\mathtt{F}'$ supérieur à $\mathtt{F}$ au sens de cette relation. Cet ajout a pour conséquence d'augmenter la flexibilité du typage proprement dit, bien qu'avec une imprécision notable.

Nous présentons un premier algorithme qui permet de calculer le type et l'effet de contrôle minimal au sens de notre spécification. Nous prouvons formellement les invariants de cet algorithme par rapport à notre spécification. Enfin, nous montrons comment étendre cette spécification de manière à autoriser un typage générique. Pour cela, nous introduisons une notion de *polymorphisme*. L'analyse de flot de contrôle présentée dans ce chapitre a été réalisée pour le compilateur Mini-FX.

Chapitre 5 : Systèmes d'Effet et Sous-Typage

Maintenant, nous remplaçons la règle d'inclusion sur les effets, présentée dans le chapitre précédent, pour l'étendre aux types. Nous définissons ainsi une relation de sous-typage contra-variante. Cette relation permet à une fonction d'avoir un type différent dans chacun de ses contextes d'appel.

Ainsi, là où la spécification du chapitre précédent n'admettait qu'une borne supérieure de l'effet pour plusieurs fonctions susceptibles d'être propagées aux mêmes points d'un programme, l'information d'effet relative à chaque fonction peut être maintenant déterminée avec plus de localité. Notre nouvelle spécification est en conséquence beaucoup plus précise qu'auparavant.

Nous présentons un nouvel algorithme permettant de déterminer le type minimal des expressions au regard de notre nouvelle spécification. Nous prouvons que cet algorithme est correct par rapport à cette spécification en utilisant des outils de preuve similaires à ceux du chapitre précédent.

Chapitre 6 : Interprétation Abstraite Séparée

Nous considérons le cadre de la compilation séparée comme la compilation des composants, ou modules, d'un programme en isolation les uns des autres. Dans le cadre d'une analyse globale de flot de contrôle, l'utilisation d'un système de type et d'effet permet au programmeur de spécifier une approximation raisonnable des propriétés statiques de chacun des modules de son programme, interfacé au reste de l'application au moyen d'une signature. Cette signature utilise un système de type et d'effet afin de donner une approximation des information de flot de contrôle pour le reste du programme.

L'interprétation abstraite séparée consiste à utiliser un système d'effet afin de spécifier le type et les effets des différents modules d'un programme, au moyen d'une interface. Cette information sert à l'interprétation abstraite de chacun des modules d'un programme (en isolation des autres) afin de donner une approximation des valeurs abstraites de leurs variables libres (définies dans d'autres modules du programme).

L'idée de base de notre contribution est de considérer que l'inférence d'effet de contrôle est cohérente avec une analyse de flot de contrôle basée sur l'interprétation abstraite (à la [Shivers91]). Dans notre système d'effet de contrôle, les types servent a représenter statiquement la structure des valeurs. Dans le cas d'une fonction, nous notons $t' \xrightarrow{d} t$ son type. L'effet latent d est ici très utile, puisqu'il donne un ensemble de fonctions pouvant correspondre à la valeur ayant ce type, mais aussi les possibles effets de contrôle correspondants.

A partir de ce type, il est alors très facile de donner une approximation de la valeur abstraite qui correspondrait à la même valeur, si nous avions utilisé l'interprétation abstraite pour la déterminer. Nous montrons que notre système d'effet de contrôle est une approximation de l'interprétation abstraite du contrôle pour le même langage et en préserve les invariants.

Nous proposons d'associer une analyse de programme basée sur l'interprétation abstraite, profitant ainsi d'une technique performante pour les expressions closes, avec l'utilisation d'un système d'effet, afin de pouvoir spécifier des informations statique en présence de compilation séparée, et donc d'information partielle sur l'ensemble d'un programme.

Chapitre 7 : Analyse d'Echappement d'Ordre Supérieur

Nous étudions dans ce chapitre l'application de l'analyse de flot de contrôle pour la *détection d'échappements*, c'est à dire la détection, dans un programme, des variables dont l'existence dépasse le cadre lexical de leur définition. Détecter cette information permet au compilateur de choisir une stratégie optimisée pour l'allocation des fonctions. Ainsi, lorsqu'une variable n'est jamais référencée en dehors du cadre lexical de sa définition, on peut alors la représenter dans la pile d'exécution. Dans le cas contraire, elle doit être représentée dans le *tas*, c'est à dire gérée par le sous-système de gestion mémoire.

Nous présentons une nouvelle technique pour l'analyse statique d'échappement, basée sur l'information que procure notre système d'effet de contrôle. Il est notable de considérer que cette analyse peut ici être mis en oeuvre en présence de fonctions d'ordre supérieur, de constructions impératives ainsi que de compilation séparée. Nous définissons une machine abstraite dotée d'une pile de contrôle pour son exécution, et nous montrons comment les variables peuvent être représentées en pile lorsque les fonctions ne les capturent pas.

Introduction

Control-flow analysis plays a key role in optimizing compilers of functional languages

Overview

Functional programming languages like Lisp [Steele90], Scheme [Rees88] and ML [Mitchell88, Appel90, Milner90] are widely recognized for their expressive power and straightforward semantics. Nevertheless they are more difficult to implement efficiently than traditional languages such as FORTRAN and C. There exists a big gap between the optimization levels of these two class of languages. Compilers of traditional languages employ a variety of interprocedural data-flow optimizations like induction-variable elimination, useless-variable elimination, constant propagation, etc [Aho86].

All these optimizations depend on the knowledge of control-flow graphs of programs at compile time. In functional languages, however, since functions are first-class values, i.e. they can be passed as parameters, returned as results of function calls or stored in memory locations, function call graphs are dynamic in nature. The lack of control-flow graphs at compile time makes interprocedural data-flow optimizations quite difficult when compiling functional languages. Moreover since higher-order functions are represented as closures including their codes and the values of their free variables, control-flow graphs help compilers optimize closure implementations [Leroy90-1, Appel89, Appel92, Tang92, Wand93], such as closure representation and closure allocation. Control-flow analysis is introduced to approximate the dynamic function call graphs at compile time. Control-flow analysis can be expressed by using the framework of *effect systems* [Tang92] and *abstract interpretation* [Shivers91].

Effect systems rely on a proof system to specify a particular analysis. This new program analysis technique separates the semantic specification from the implementation of the analysis. Thanks to such a separation, static analyses based on effect systems can be easily extended to other language features and used to formally specify optimization techniques. However abstract interpretation does not provide such a clear separation. As a consequence, optimization techniques related to an analyzer based on abstract interpretation are hard to formalize and extend. Another advantage of effect systems over abstract interpretation is that they can be straightforwardly extended in the context of separate compilation via module signatures. However the abstract interpretation approach performs more precise control-flow analysis thanks to fixed point iteration techniques. The precision of control-flow analysis controls the precision of the interprocedural optimizations performed by optimizing compilers. [Shivers91] introduces the notion of *order* to mesure the precision of control-flow analysis. The

order indicates the number of pending calls remembered during the analysis of a given application expression. My thesis presents zeroth and first-order control-flow analysis systems expressed by both *effect systems* and *abstract interpretation* and studies the application of control-flow information in optimizing closure allocation.

Effect systems [Lucassen87, Lucassen88] extend classical polymorphic type systems [Milner78, Tofte87] with effect information. Just like types describe the possible values of expressions, effects specify their possible evaluation behaviors. However the introduction of effects imposes new constraint on the typability of languages. The existing effect systems [Talpin92-1, Dornic91, Tang92] use the notion of *subeffecting* to increase the flexibility of effect systems. It allows expressions to admit larger effects whenever effect matching would cause type clashes, thus limiting the accuracy of effect systems. Instead of subeffecting, my thesis introduces the notion of *subtyping* [Tang93]. The subtype relation is limited to the subsumption relation on effect information. Subtyping allows the same function to have different supertypes at different call sites, as long as certain they satisfiy certain subtype relations. The introduction of subtyping improves both flexibility and accuracy of effect systems. Control-flow analysis by effect systems can be performed in presence of higher-order functions, imperative constructs and separate compilation. However it collapses different call contexts together, thus limiting the analysis accuracy to the 0th-order control-flow analysis (0CFA).

Abstract interpretation [Cousot77, Cousot79, Mycroft81] is another static analysis framework. It approximates language semantics by using the fixed point iteration technique. Control-flow analysis based on abstract interpretation [Shivers91] can distinguish different call contexts, thus performing nth-order control-flow analysis (nCFA), but fails to support separate compilation. If the abstract interpretation approach performs more precise static analysis due to fixed point iteration techniques, effect systems support separate compilation more naturally via module signatures [MacQueen90, Sheldon90]. My thesis introduces the new notion of *separate abstract interpretation* [Tang94] that combines effect systems and abstract interpretation in a single framework. It extends abstract interpretation in the context of separate compilation based on the type and effect information of module signatures. The control-flow analysis expressed by the separate abstract interpretation is as precise as the abstract interpretation approach on closed expressions, but is also able to tackle expressions with free variables by using their type and control-flow information to approximate their abstract values.

As a direct application of control-flow analysis, my thesis presents a new escape analysis for optimizing closure allocation [Tang92]. Escape analysis identifies the free variables that outlive their lexical scope in function definitions. This compile-time knowledge of *escaping variables* helps compilers choose a more efficient allocation strategy for closures, i.e. non-escaping variables can be safely stored in the stack, while heap allocation is only used for escaping ones.

Outline

The thesis presents different control-flow analysis systems expressed by effect systems (Chapter 3, 4, 5) and abstract interpretation (Chapter 6). The related work is discussed in each chapter and the proofs are presented in the appendix at the end of the thesis.

- Chapter 2 gives an informal description of what effect systems and abstract interpretation are. We discuss the basic ideas, describe their frameworks for static analysis and present their applications in several static analyses.

- Chapter 3 presents a simple control-flow effect system. We give our language syntax, define the dynamic semantics, introduce the static semantics for control-flow analysis and state its consistency w.r.t. the dynamic semantics.

- Chapter 4 introduces subeffecting in the static semantics to increase the flexibility of effect systems. We present a reconstruction algorithm $\mathcal{R}$ that reconstructs type and control-flow information of expressions based on subeffecting and prove it sound and complete w.r.t. the static semantics.

- Chapter 5 introduces subtyping in the static semantics to increase the accuracy of the subeffecting effect systems. We present a reconstruction algorithm $\mathcal{S}$ that reconstructs type and control-flow information of expressions based on subtyping and prove it sound and complete w.r.t. the static semantics.

- Chapter 6 introduces the new technique of separate abstract interpretation to perform control-flow analysis. We describe the abstract interpretation semantics and state its consistency with the type semantics, show the approach of approximating abstract values of free variables by their types and control-flow information and prove that separate abstract interpretation is a conservative extension of abstract interpretation.

- Chapter 7 describes a new escape analysis based on the control-flow effect systems. We present an algorithm $\mathcal{I}$ to identify escaping variables of functions, introduce an efficient closure allocation strategy, describe a stack-based abstract machine and compare our analysis with other escape analyses, particularly that based on abstract interpretation [Goldberg90].

- Finally, we conclude and discuss future work.

Chapter 2

Static Analysis Approaches

Effect systems and abstract interpretation provide general frameworks of static analyses.

Modern compilers perform a variety of program analyses in order to produce good code [Steele78, Cardelli84, Appel87, Kelsey89, Leroy90-2]. The goal of the analyses is to approximate at compile time evaluation behaviors of programs, which help compilers identify optimization opportunities. Standard analysis techniques have been developed [Aho86] for the traditional languages like PASCAL and C, which are built upon static control-flow graphs. However the traditional analysis approaches are not applicable to higher-order programming languages, since control-flow graphs are absent at compile time. Therefore *effect systems* and *abstract interpretation* are introduced to perform static analysis for the programming languages in which functions are first-class values.

2.1 Effect Systems

Type systems have been developed in both traditional and functional languages [Milner78, MacCracken79, Damas82, Cardelli85, Tofte90, O'Toole90]. By using types to describe the data structure of values, programmers can describe the intended specifications of expressions, which makes programs more structured, and compilers can detect type errors, which allows greater execution-time efficiency. The static strong typing of programming languages requires that the type of every expression can be determined at compile time. It allows type inconsistencies to be discovered at compile time and guarantees that executed programs are type consistent. Static typing is a popular technique used for program analysis. Nevertheless by imposing constraints on acceptable expressions, static type systems, may lead to the loss of flexibility and expressive power of programming languages. Polymorphism, expressed by both generic types [Milner78, Kanellakis89, Leroy91] and subtyping [Cardelli88, Aiken93, Fuh88, Stansifer88, Benjamin92] is often introduced to allow an expression to have more than one type.

Effect systems extend classical type systems with effect information. Just like types describe the possible values of expressions, effects specify their possible evaluation

properties. In effect systems, a classical function type $t' \rightarrow t$ [1] is extended to $t' \xrightarrow{\mathtt{F}} t$ where the *latent effect* $\mathtt{F}$ records the evaluation property of functions of this type. Since the latent effect statically links a function from its definition site to its call sites, it plays an important role in effect systems. The introduction of effects increases the applicability of type systems. Nevertheless since effects have their own domains and richer algebras than types, they make type systems much more complicated.

2.1.1 Effect Semantics

Types in effect systems are extended with effect information. A type t can either be a basic type like *int*, *bool* or *unit*, a reference type $ref(t)$ that represents updatable memory locations containing values of the type t, or a function type $t' \xrightarrow{\mathtt{F}} t$ representing a function that accepts arguments of the type t' and return values of the type t. The evaluation property of the function body is approximated by the latent effect $\mathtt{F}$.

$$t \in \quad \text{Type} \quad = \mathit{BasicTypes} \mid ref(t) \mid t' \xrightarrow{\mathtt{F}} t$$

Just as types, effects have their own domain and algebra corresponding to the evaluation properties we are interested in. For example, in a memory side-effect analysis, an effect is defined as a set of store operations and belongs to a set algebra. An effect $\mathtt{F}$ can be defined as either the emptyset $\emptyset$ meaning the absence of side-effects, a basic store operation like `init`, `read` or `write`, or a set of side-effects gathered with the infix union operator $\cup$:

$$\begin{array}{rl} \mathtt{F} ::= & \emptyset \\ & \mathtt{read} \mid \mathtt{write} \mid \mathtt{init} \\ & \mathtt{F} \cup \mathtt{F}' \end{array}$$

The side-effects belong to the following set algebra :

$$\begin{array}{lcl} \mathtt{F} \cup (\mathtt{F}' \cup \mathtt{F}'') & = & (\mathtt{F} \cup \mathtt{F}') \cup \mathtt{F}'' \\ \mathtt{F} \cup \mathtt{F}' & = & \mathtt{F}' \cup \mathtt{F} \\ \mathtt{F} \cup \emptyset & = & \mathtt{F} \\ \mathtt{F} \cup \mathtt{F} & = & \mathtt{F} \end{array}$$

An effect system is uniformly specified by a set of inference rules [Plotkin81]. An inference rule (called *name*) is made of a set of predicates $\mathtt{P}$ and $\mathtt{P}_1 \ldots \mathtt{P}_n$, which means that the conclusion $\mathtt{P}$ holds if the premises $\mathtt{P}_1 \ldots \mathtt{P}_n$ have been proved. We note:

$$(name) : \frac{\mathtt{P}_1, \ldots \mathtt{P}_n}{\mathtt{P}}$$

If the conclusion $\mathtt{P}$ always holds, the inference rule is degenerated to an axiom.

$$(name) : \mathtt{P}$$

In effect semantics, a predicate $\mathcal{E} \vdash \mathtt{e} : \mathtt{T}, \mathtt{F}$ means that in an environment $\mathcal{E}$, the expression $\mathtt{e}$ is evaluated to a value of the type $\mathtt{T}$ and its possible evaluation behavior

[1] The whole syntax is formally presented in Chapter 5

is recorded by the effect F. Note that the environment $\mathcal{E}$ defines the types of the free variables in the expression e.

Using inference rules, the effect semantics inductively specifies the predicate for each expression on its structure. The crucial rules are the (*abs*) rule for lambda abstractions and (*app*) rule for applications.

Given a type environment $\mathcal{E}$ and a lambda abstraction (λ (x) e), x is of the type t' the function body e is evaluated to a value of the type t in the extended type environment $\mathcal{E}[\mathtt{x} \mapsto t']$ and the evaluation property of e is approximated by the effect F. Then the lambda abstraction has the function type $t' \xrightarrow{\mathtt{F}} t$. The latent effect is thus introduced in the function type.

$$(abs): \frac{\mathcal{E}[\mathtt{x} \mapsto t'] \vdash \mathtt{e} : t, \mathtt{F}}{\mathcal{E} \vdash (\lambda\ (\mathtt{x})\ \mathtt{e}) : t' \xrightarrow{\mathtt{F}} t, \emptyset}$$

When such a function is applied in the application case, its latent effect appears in the resulted effect of the application expression to approximate the evaluation property of the function body. Given a type environment $\mathcal{E}$ and an application (e e′), if e and e′ are evaluated to values of the type $t' \xrightarrow{\mathtt{F}''} t$ and t', and their evaluation behaviors are approximated by the effects F and F′ respectively, then (e e′) is of the type t and its evaluation behavior is approximated by $\mathtt{F} \cup \mathtt{F}' \cup \mathtt{F}''$ where F″ represents the evaluation behavior of the function body.

$$(app): \frac{\begin{array}{l}\mathcal{E} \vdash \mathtt{e} : t' \xrightarrow{\mathtt{F}''} t, \mathtt{F} \\ \mathcal{E} \vdash \mathtt{e}' : t', \mathtt{F}'\end{array}}{\mathcal{E} \vdash (\mathtt{e}\ \mathtt{e}') : t, \mathtt{F} \cup \mathtt{F}' \cup \mathtt{F}''}$$

Note that the latent effect statically links functions from their definition sites to their call sites, thus playing a key role in effect systems.

2.1.2 Verification and Inference

There are two families of effect systems based on either verification or inference. Effect verification systems ask programmers to explicitly specify the types and effects of some expressions in a program, and statically verify the type and effect correctness of the program based a type checking technique. The effect system presented in [Gifford87, Lucassen87, Hammel88] defines the FX programming language and uses an effect verification system [Jouvelot88] to check polymorphic types and side-effect declarations in FX programs.

In order to spare programmers from specifying type and effect information, [Jouvelot91, Talpin92-1] introduce a new effect system that automatically determines the types and effects for implicitly typed programs. This effect inference system extends classical type inference systems [Milner78, Tofte87] with effect information and introduces the notion of *algebraic reconstruction* and considers the type and effect inference issue as a constraint satisfaction problem.

The difference of these two sorts of effect systems is represented by the (*abs*) rule for the lambda abstraction. In inference systems, the type of the parameter of each

lambda abstraction is computed automatically by a reconstruction algorithm, while in verification systems, it has to be explicitly specified by programmers like below :

$$(abs): \frac{\mathcal{E}[\mathtt{x} \mapsto t'] \vdash \mathtt{e} : t, \mathtt{F}}{\mathcal{E} \vdash (\lambda\ (\mathtt{x} : t')\ \mathtt{e}) : t' \xrightarrow{\mathtt{F}} t, \emptyset}$$

2.1.3 Analysis Framework

Effect systems provide a general framework for performing static analysis of programs. Suppose we have a program written in some programming language, and we wish to approximate at compile time some property `X` about the program evaluation process. For instance, `X` can be the set of side-effects performed during the evaluation of the program, or the time that the programs need to run, etc. We use the following three-step process to obtain the approximation of the property `X`.

1. We start with a *dynamic semantics* for the language that precisely defines what the program means. The dynamic semantics defines the result value `V` and the property `X` of the program.

2. Then, we develop a *static semantics* that conservatively approximates the dynamic semantics. The static semantics defines the type `T` and effect `F` of the program where the type `T` describes the data structure of the result value `V` , and the effect `F` is an approximation of the property `X`.

3. The static semantics defines the static analysis we wish to perform. However the semantic specification is too abstract to tell how to compute the static information. The final step is to define a *reconstruction algorithm* that computes the type $\mathtt{T}'$ and the effect $\mathtt{F}'$ of the program.

To guarantee that the static information obtained by this effect system is a conservative approximation of the program property, we have to prove the following correctness results :

- The static semantics must be consistent with the dynamic semantics, which means that the effect `F` defined by the static semantics is a conservative approximation of the property `X` defined by the dynamic semantics.

- The reconstruction algorithm must be sound and complete w.r.t. the static semantics, stating that, for each program, the type $\mathtt{T}'$ and the effect $\mathtt{F}'$ computed by the reconstruction algorithm satisfy the static semantics and cover all types `T` and effects `F` derivable by the static semantics.

2.1.4 Applications

Effect systems have been used to perform several static analyses [Jouvelot88, Talpin92-1, Dornic91, Tang92, Consel93]. Effect systems have the following properties : (1) latent effects are introduced in function types to approximate the evaluation property of functions; (2) Memory locations are explicitly expressed by reference types $ref(t)$, which makes them clearly manifest at compile time ; (3) Types and effects can be expressed in

module signatures. Thanks to these properties, static analysis based on effect systems can be performed in the presence of higher-order functions, imperative constructs and separate compilation. Here we give some examples of static analyses performed by effect systems.

- Side-effects analysis

  [Talpin92-1] introduces a static side-effect analysis system based on a type and effect inference system that approximates the memory operations like read, write and initialize. The notion of *region* [Gifford87] is used to specify possible memory sharing. [Talpin92-2] uses an observation criterion that discards the side-effects related to local data structures, thus allowing more precise side-effect information. The compile-time knowledge of side-effects is important to parallel code generation. A prototype compiler [Talpin93-1] has been designed that targets the FX programming language to the Connection Machine.

- Complexity analysis

  [Dornic91] suggests an effect system to estimate the worst-case evaluation time of expressions. Even if it fails to accurately approximate recursive functions, it is helpful for choosing an efficient load balance strategy when compiling programs for multiprocessors.

- Binding-time analysis

  [Consel93] uses an effect system to determine which variables can be bound to their values at compile time. This binding-time information is of importance when performing partial evaluation or constant folding on programs.

- Control-flow analysis

  [Tang92] applies effect systems for approximating dynamic function call graphs of functional languages at compile time. By using the type and control-flow information, an escape analysis is performed to statically identify the free variables of functions that outlive their definition scopes, which helps compilers choose an efficient closure allocation strategy. [Tang93] introduces subtyping in effect systems to avoid effect information to be merged together, thus improving both flexibility and accuracy of effect systems. [Tang94] introduces a more precise control-flow analysis by extending abstract interpretation to support separate compilation based on type and control-flow information. These control-flow analyses form the core of this thesis.

2.2 Abstract Interpretation

Abstract Interpretation [Cousot77, Cousot79, Mycroft81] is a theory of semantics approximation which provides a powerful approach for static analysis [Mycroft81, Deutsch90, Goldberg90, Shivers91]. It is built upon the denotational semantics by approximating the fixpoint nature of the language semantics. Denotational semantics was developed to define the formal semantic descriptions of programming languages. It uses semantic

evaluation functions which map syntactic constructs in programs to the abstract values which they denote. Since these evaluation functions are usually recursively defined, they may or may not suggest a way of implementing the language.

The classical framework starts from a standard denotational semantics describing the evaluation process of a programming language. Then an exact semantics is designed to precisely describe the program properties. The exact semantics is often described using fixpoints on ordered structures. Since the exact semantics is not always computable, an abstract semantics is designed to approximate it. The connection between the static and the abstract semantics is specified by abstraction functions, which map exact values to their abstract counterparts. The definition of the abstract function is based on the fixpoint approximation. The static fixpoint approximation approach simplifies equations of the exact semantics to abstract ones that approximate the exact semantics.

Since the abstract interpretation approach is built on the fixpoint approximation of the exact semantic, which requires the whole structure of programs, while effect systems use an unification-based reconstruction algorithm that only relies on the local structure of program syntax, abstract interpretation allows more precise static analysis than effect systems. Nevertheless type and effect information can be easily specified by module signatures, so that effect systems support separate compilation more naturally.

2.2.1 Analysis Framework

Abstract interpretation provides a general framework for static analysis. Suppose we have a program written in some programming language and we wish to approximate at compile time some property `X` about the program evaluation process. We use the following three-step procedure to obtain the approximation of the property `X`.

1. We start with a *standard denotational semantics* for the language that precisely defines what the program `P` means. The standard denotational semantics specifies the result value `V` of the program.

2. Then, we develop an *exact semantics* that precisely expresses the property `X` of the program. We derive this exact semantics from the original standard semantics. So, if the standard denotational semantics specifies the evaluation result `V` of the program, then the exact semantics describes its evaluation property `X`, which constitutes a precise, formal definition of the property we want to analyze.

3. Since the precision of the exact semantics typically implies that it may be uncomputable at compile time, an *abstract semantics* is defined to approximate the exact semantics. The abstract semantics defines the approximation $\hat{\mathtt{X}}$ of the evaluation property of the program by trading accuracy for compile-time computability.

The correctness of this abstract interpretation approach requires the following correctness results :

- The equations defining the exact and abstract semantics have solutions.
- The abstract semantics safely approximates the exact semantics.
- The abstract semantics is computable.

2.2.2 Applications

Abstract interpretation has been used for both imperative [Cousot77, Cousot79] and higher-order languages [Mycroft81, Deutsch90, Shivers91]. It provides a basic framework for performing static analysis of programming languages in presence of higher-order functions and side-effects.

- Strictness analysis

  [Mycroft81] extends the idea of abstract semantics to a functional language and uses a complex semantic structure called a powerdomain. He applies the abstract interpretation framework to perform a static strictness analysis for normal-order languages that identifies opportunities to evaluate function arguments with the call-by value rule instead of the call-by need rule without changing the result of the program.

- Escape analysis

  [Goldberg90] presents an application of abstract interpretation for escape analysis, which helps optimize the allocation of closures. An interesting point of this analysis is that, for any function, its type can be used to approximate its arguments that cause the greatest escapement possible.

- Control-flow analysis

  [Shivers91] uses abstract interpretation to approximate control-flow graphs of a higher-order language allowing side-effects. Programs are transformed to CPS form (Continuation-Passing Style) so that transfers of control are uniformly represented as tail-recursive function calls. This abstract interpretation approach can perform nth-order control-flow analysis, but fails to support separate compilation.

Chapter 3

Control-Flow Effect System

Control-flow analysis can be expressed by effect systems.

3.1 Introduction

Control-flow analysis strives to approximate dynamic control-flow behaviors of functional languages at compile time. Control-flow information can be defined as the set of function names *possibly* called during the evaluation of expressions. This chapter presents a new control-flow analysis based on effect systems. Effect systems extend classical type systems with effect information. By using effects to specify program evaluation behaviors, effect systems provide a powerful method to perform static analysis in the presence of higher-order functions, imperative constructs and separate compilation.

In the sequel, we present our language syntax (Section 3.2), describe the dynamic semantics (Section 3.3), define a simple static semantics of control-flow analysis (Section 3.4) and state its consistency with respect to the dynamic semantics (Section 3.5). Finally we describe related work (Section 3.6) before concluding (Section 3.7). All proofs are presented in Appendix 1.

3.2 Language Definition

My thesis focuses on a simple functional language for the simplicity of presentation. Nevertheless it could be extended to more complicated languages including imperative constructs, let-bindings, module constructs, etc. Possible extensions are discussed in each chapter.

$\mathtt{e} ::=$	$\mathtt{x}$	*value identifier*
	$(\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e})$	*abstraction*
	$(\mathtt{rec}_{\mathtt{n}}\ (\mathtt{f}\ \mathtt{x})\ \mathtt{e})$	*recursive function*
	$(\mathtt{e}\ \mathtt{e}')_{\mathtt{l}}$	*application*

Note that function definitions and function calls are tagged with unique labels (n and l) that allow to uniquely distinguish them.

$$\begin{array}{lll} \mathtt{n} \in \text{LFun} & = \text{Label} & \textit{Label of functions} \\ \mathtt{l} \in \text{LCall} & = \text{Label} & \textit{Label of function calls} \end{array}$$

How this labeling is actually performed is not important, as long as the uniqueness property is preserved. However these labels will appear in types and, eventually, module type signatures, so they must be easily understandable by the user. For instance, the label of a function could consist of an identifier (indicating the name of the module where it is defined) and a number (distinguishing it from other functions in the same module)

3.3 Dynamic Semantics

The dynamic semantics not only defines the values of expressions, but also keeps track of control-flow traces occurring during their evaluation.

A computable value v is either an integer i, or a closure. The closure cl is composed of the function name, the argument name, the body expression and the lexical environment in which the function is defined. The environment E is a finite map from identifiers to values. The control-flow behavior b of an expression is a set of function names that are called during its evaluation. The empty set indicates the absence of control-flow traces.

$$\begin{array}{llll} v \in & \text{Value} & = \text{Int} + \text{Closure} & \textit{value} \\ cl \in & \text{Closure} & = \text{LFun} * \text{Id} * \text{Exp} * \text{Env} & \textit{closure} \\ E \in & \text{Env} & = \text{Id} \mapsto \text{Value} & \textit{environment} \\ f \in & \text{Trace} & = \mathcal{P}(\text{LFun}) & \textit{control-flow trace} \end{array}$$

The dynamic semantics is specified by a set of inference rules [Plotkin81]. Given an environment E , it associates an expression $\mathtt{e}$ with the value v it computes and the set of function names b called during its evaluation. We note :

$$E \vdash \mathtt{e} \rightarrow v, b$$

$$(var) : \frac{\mathtt{x} \in Dom(E)}{E \vdash \mathtt{x} \rightarrow E(\mathtt{x}), \emptyset}$$

$$(abs) : E \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) \rightarrow (\mathtt{n}, \mathtt{x}, \mathtt{e}, E_{\mathtt{x}}), \emptyset$$

$$(rec) : \frac{cl = (\mathtt{n}, \mathtt{x}, \mathtt{e}, E[\mathtt{f} \mapsto cl])}{E \vdash (\mathtt{rec}_{\mathtt{n}}\ (\mathtt{f}\ \mathtt{x})\ \mathtt{e}) \rightarrow cl, \emptyset}$$

$$(app) : \frac{\begin{array}{l} E \vdash \mathtt{e} \rightarrow (\mathtt{n}, \mathtt{x}, \mathtt{e}'', E'), b \\ E \vdash \mathtt{e}' \rightarrow v', b' \\ E'[\mathtt{x} \mapsto v'] \vdash \mathtt{e}'' \rightarrow v, b'' \end{array}}{E \vdash (\mathtt{e}\ \mathtt{e}') \rightarrow v, b \cup b' \cup b'' \cup \{\mathtt{n}\}}$$

where for any function f, $f[\mathtt{x} \mapsto v]$ is the extension of f with the property that $f[\mathtt{x} \mapsto v](\mathtt{x}) = v$ and $f[\mathtt{x} \mapsto v](\mathtt{y}) = f(\mathtt{y})$, $Dom(f)$ is the domain of f and $f_{\mathtt{x}}$ is f in which $\mathtt{x}$ is unbound.

We use a simple example `demo` to show how control transfers during the evaluation of the program.

$$\begin{array}{l}\mathtt{demo}=((\lambda_{\mathtt{n}_f}\ \ (\mathtt{f}) \\ \qquad\qquad (+\ (\mathtt{f}\ (\lambda_{\mathtt{n}_a}\ (\mathtt{a})\ \mathtt{a}))_{\mathtt{l}_a} \\ \qquad\qquad\quad (\mathtt{f}\ (\lambda_{\mathtt{n}_b}\ (\mathtt{b})\ \mathtt{b}))_{\mathtt{l}_b})) \\ \qquad (\lambda_{\mathtt{n}_g}\ (\mathtt{g})\ (\mathtt{g}\ 1)_{\mathtt{l}_g}))_{\mathtt{l}_f}\end{array}$$

where the lambda expression $\mathtt{n}_g$ is bound to the variable `f` at the call site $\mathtt{l}_f$ and is applied to the arguments $(\lambda_{\mathtt{n}_a}\ (\mathtt{a})\ \mathtt{a})$ and $(\lambda_{\mathtt{n}_b}\ (\mathtt{b})\ \mathtt{b})$ at the call sites $\mathtt{l}_a$ and $\mathtt{l}_b$.

At the call site $\mathtt{l}_a$, the function $\mathtt{n}_g$ (via `f`) is called by binding the identity function $\mathtt{n}_a$ to the variable `g`; then control transfers to the function body of $\mathtt{n}_g$, where $\mathtt{n}_a$ (via `g`) is applied to the argument 1 at $\mathtt{l}_g$; then control transfers to the function body of $\mathtt{n}_a$, where there is no function calls, i.e. its call set is $\emptyset$. Therefore, the control-flow traces of $(\mathtt{f}\ (\lambda_{\mathtt{n_a}}\ (\mathtt{a})\ \mathtt{a}))$ is recorded by the call set $\{\mathtt{n}_g, \mathtt{n}_a\}$ meaning that during the evaluation of this function application, the functions $\mathtt{n}_g$ and $\mathtt{n}_a$ are called. Similarly with the application $(\mathtt{f}\ (\lambda_{\mathtt{n_b}}\ (\mathtt{b})\ \mathtt{b}))$, its call set is $\{\mathtt{n}_g, \mathtt{n}_b\}$.

In functional languages, since functions are first-class values, control-flow traces defined by the dynamic semantics can not be determined at compile time. Therefore we define a static semantics to approximate the dynamic one.

3.4 Static Semantics

For each expression the static semantics specifies its type and a set of functions *possibly* called during its evaluation.

The control-flow effect c abstracts the trace b in the dynamic semantics and thus records all functions possibly called during the evaluation of an expression. A control-flow effect c can either be a constant $\emptyset$ meaning the absence of function calls, a singleton $\{\mathtt{n}\}$ where `n` is a function name, or a set of function names indicated by the infix union operator $\cup$. A type t can either be the basic type int, or a function type $t' \xrightarrow{c} t$ where the latent control-flow effect c records the set of functions possibly called when a function of this type is called. A type environment $\mathcal{E}$ is a finite map from identifiers to types.

$c \in$	Control	$c ::$	$= \emptyset \mid \{\mathtt{n}\} \mid c \cup c'$	*control-flow effect*
$t \in$	Type	$t ::$	$= int \mid t' \xrightarrow{c} t$	*type*
$\mathcal{E} \in$	TEnv		$=$ Id $\mapsto$ Type	*type environment*

The static semantics is specified by a set of inference rules. Given a type environment $\mathcal{E}$, it associates an expression `e` with its type t and control-flow effect c. We note :

$$\mathcal{E} \vdash \mathtt{e} : t, c$$

The latent control-flow effect is introduced in function types by the (*abs*) rule for lambda abstraction and used to approximate control-flow traces of function bodies in the (*app*) rule for function application. In the abstraction case, the current function

name is added to the control-flow effect of the lambda body; the resulting set is the latent control-flow effect of the lambda expression. When such a function is applied, in the function application, this latent control-flow information is used to determine the functions possibly called while evaluating the function body.

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

$$(abs): \frac{\mathcal{E}[\mathtt{x} \mapsto t'] \vdash \mathtt{e} : t, c}{\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t' \stackrel{\{\mathtt{n}\} \cup c}{\rightarrow} t, \emptyset}$$

$$(rec): \frac{\mathcal{E}[\mathtt{f} \mapsto t] \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t, \emptyset}{\mathcal{E} \vdash (\mathtt{rec}_{\mathtt{n}}\ (\mathtt{f}\ \mathtt{x})\ \mathtt{e}) : t, \emptyset}$$

$$(app): \frac{\begin{array}{c}\mathcal{E} \vdash \mathtt{e} : t' \stackrel{c''}{\rightarrow} t, c \\ \mathcal{E} \vdash \mathtt{e}' : t', c'\end{array}}{\mathcal{E} \vdash (\mathtt{e}\ \mathtt{e}') : t, c \cup c' \cup c''}$$

3.5 Consistency Theorem

We use the proof method introduced in [Tofte87] to show that the static and dynamic semantics are consistent with respect to a structural relation between values and types, defined as the maximal fixed point of a monotonic function.

3.5.1 Definition

We introduce ":" to define the consistency relation between values and types, noted as $v : t$. This can be easily extended to their environments.

Definition 3.1 (Types of Values and Environments)

$$\begin{array}{l} i : int \\ (\mathtt{n}, \mathtt{x}, \mathtt{e}, E) : t \Leftrightarrow \exists \mathcal{E},\ \text{s.t. } E : \mathcal{E} \quad and \quad \mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t \\ E : \mathcal{E} \quad \Leftrightarrow \quad \forall \mathtt{x} \in Dom(E),\ \mathtt{x} \in Dom(\mathcal{E}) \quad and \quad E(\mathtt{x}) : \mathcal{E}(\mathtt{x}) \end{array}$$

The above structural property does not uniquely define a relation between values and types and must be regarded as a fixed point equation on the domain $\mathcal{R} = \text{Value} * \text{Type}$. We define a function $\mathcal{F}$ on $\mathcal{P}(\mathcal{R}) \mapsto \mathcal{P}(\mathcal{R})$, which for any $\mathcal{Q} \subseteq \mathcal{R}$, satisfies the following definition :

$$\begin{array}{l} \mathcal{F}(\mathcal{Q}) = \{(v, t) \in \mathcal{Q} \mid \\ \quad if\ \ v = i \qquad\qquad then\ t = int \\ \quad if\ \ v = (\mathtt{n}, \mathtt{x}, \mathtt{e}, E) \quad then\ \exists \mathcal{E}, s.t.\ \forall \mathtt{x} \in Dom(E) \\ \qquad\qquad\qquad\qquad \mathtt{x} \in Dom(\mathcal{E})\ and\ (E(\mathtt{x}), \mathcal{E}(\mathtt{x})) \in \mathcal{Q}\ and\ \mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t\} \end{array}$$

Its fixed points are the relations on $\mathcal{R}$ that verify the structural property defined by Definition 3.1. In order to guarantee the existence of the fixed points of $\mathcal{F}$, it is sufficient to show that $\mathcal{F}$ is monotonic [Stoy77].

Lemma 3.1 (Monotony of $\mathcal{F}$) *If $\mathcal{Q}$ and $\mathcal{Q}'$ are two subsets of the domain $\mathcal{R}$.*

$$\mathcal{Q} \subseteq \mathcal{Q}' \Rightarrow \mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$$

3.5.2 Fixpoints

Since the function $\mathcal{F}$ is monotonic, it has a minimal and a maximal fixpoint in the complete lattice $(\mathcal{P}(\mathcal{R}), \subseteq)$, namely

$$\mathit{lfp}(\mathcal{F}) = \cap\{\mathcal{Q} \subseteq \mathcal{R} \mid \mathcal{F}(\mathcal{Q}) \subseteq \mathcal{Q}\}$$

and

$$\mathit{gfp}(\mathcal{F}) = \cup\{\mathcal{Q} \subseteq \mathcal{R} \mid \mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})\}$$

To reach the least fixpoint of $\mathcal{F}$, we start from a bottom set $\mathcal{Q} = \emptyset$ and gradually construct its least fixpoint $\mathit{lfp}(\mathcal{F})$. The set $\mathcal{F}(\emptyset)$ consists of those relations that can be proved without reference to any other relations. For example we have $(1, \mathit{int}) \in \mathcal{F}(\emptyset)$ for the $\mathcal{F}$ we have defined. Next, $\mathcal{F}(\mathcal{F}(\emptyset))$ contains the relations that can be proved based on the relations already proved by $\mathcal{F}(\emptyset)$ and so on. Since $\mathcal{F}$ is monotonic, the limit of the chain

$$\emptyset \subseteq \mathcal{F}(\emptyset) \subseteq \mathcal{F}(\mathcal{F}(\emptyset)) \ldots$$

is the least fixpoint of $\mathcal{F}$. For any relation q, $q \in \mathit{lfp}(\mathcal{F})$ if and only if q can be proved true by $\mathcal{F}$.

The maximal fixpoint finding begins with a top set and gradually eliminates those relations that have been proved wrong. Starting from $\mathcal{Q} = \mathcal{R}$, meaning that at the outset $\mathcal{F}$ rejects nothing, $\mathcal{F}(\mathcal{R})$ is the set of relations that cannot be rejected based on the relation in $\mathcal{R}$. By using the monotonic $\mathcal{F}$ we have

$$\ldots\mathcal{F}(\mathcal{F}(\mathcal{R})) \subseteq \mathcal{F}(\mathcal{R}) \subseteq \mathcal{R}$$

and the limit of this chain is the maximal fixpoint of $\mathcal{F}$. For any relation q, $q \in \mathit{gfp}(\mathcal{F})$ if and only if $\mathcal{F}$ can never prove q to be wrong.

Maximal fixpoints are of more interest whenever we define consistency relations, since in some cases such as in languages with side-effects, the least fixpoint cannot be constructed (see [Tofte87]) and the minimal fixpoint $\mathit{lfp}(\mathcal{F})$ is strictly contained in the maximal fixpoint $\mathit{gfp}(\mathcal{F})$. Therefore we choose the maximal fixpoint of the function $\mathcal{F}$ to define the relation between types and values.

Definition 3.2 (Types of Values)

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

The subset operation $\subseteq$ is used to define the consistency between dynamic and static control-flow information.

Using these definitions, we can express that the static semantics conservatively approximates the dynamic semantics.

Theorem 3.1 (Consistency of Static Semantics)

$$\left.\begin{array}{l} E \vdash \mathsf{e} \rightarrow v, b \\ \mathcal{E} \vdash \mathsf{e} : t, c \\ E : \mathcal{E} \end{array}\right\} \;\Rightarrow\; \left\{\begin{array}{l} v : t \\ b \subseteq c \end{array}\right.$$

Proof *See Appendix 1*

3.6 Related Work

Shivers's thesis [Shivers91] presented a control-flow analysis based on abstract interpretation. This analysis is more precise than the one presented here. It can distinguish certain call contexts, but fails to support separate compilation, thus limiting its pratical application. By contrast, our control-flow analysis is performed by a type and effect system that supports the separate compilation of modules, but collapses all call contexts together.

Control effects, defined in [Jouvelot89], are somewhat related to the control-flow information we gather here. However, these control effects are targeted to non-functional behaviors, such as those created by branches or continuations. Also, this analysis is targeted to an explicitly typed language, which allows explicit polymorphism.

The flow analysis described in [Bondorf93], which is used in their binding-time analysis system, traces the flow of function values in an untyped language. The only information gathered there for first-class functions is the possible arities of functions reaching a given program point. Since we restrict ourselves to typed languages, this information is a special case of our type analysis. Their flow constraint set is generated by a one-pass compositional run over programs and solved by a rewriting system, which has the same complexity as our type and effect inference system.

Effect systems have been used to approximate program evaluation behaviors in higher-order language, such as side-effects [Talpin92-1], evaluation complexity [Dornic91], etc. Control-flow analysis is another application of effect systems.

3.7 Conclusion

We presented a new control-flow analysis system based on effect systems. This control-flow effect system can be used in presence of higher-order functions, imperative constructs and separate compilation. We defined the dynamic semantics of the language, described a static semantics for control-flow analysis, and stated its consistency w.r.t. to the dynamic semantics.

If this simple static semantics is useful to lay the ground to control-flow analysis by effect systems, it suffers from shortcomings that are going to be presented and dealt with in Chapter 4,5. Thus even though a reconstruction algorithm could easily be designed for this semantics, we will delay this aspect until the next chapters.

Chapter 4

Subeffecting Effect Systems

Subeffecting improves flexibility of effect systems but at the price of their accuracy.

4.1 Introduction

We extend the static semantics defined in Chapter 3 with the subeffecting rule. Subeffecting allows expressions to admit larger effects when a type mismatch occurs, thus improving the flexibility of the effect system. However it forces a function to have a unique type in different call contexts by merging their effects together, thus limiting the accuracy of the effect system. We introduce a reconstruction algorithm that for each expression computes its type and control-flow effect based on subeffecting. The reconstruction algorithm is sound and complete w.r.t. the static semantics.

We introduce the subeffecting rule (Section 4.2), present the corresponding reconstruction algorithm and prove it sound and complete w.r.t. the static semantics (Section 4.3). Finally we describe system extensions (Section 4.4) before concluding (Section 4.5). All proofs are presented in Appendix 2.

4.2 The Subeffecting Rule

The effect system defined in Chapter 3 extends classical type systems with effect information. However the interplay of types and effects introduces new *constraint* on the typability of expressions. Effect checking may force the rejection of programs that would be type checked if no effects were present, thus reducing the flexibility of effect systems. We use the example `demo` to show this problem. For clarity, we use i to indicates the type *int*.

In classical type systems, since the lambda expressions $(\lambda_{\mathtt{n}_a}\ (\mathtt{a})\ \mathtt{a})$ and $(\lambda_{\mathtt{n}_b}\ (\mathtt{b})\ \mathtt{b})$ have the same type $i \to i$, the variable `f` admits the same type $(i \to i) \to i$ in its three instances. Therefore this program can be type checked.

In the previously defined effect system, due to the introduction of the latent control-flow effects $\{\mathtt{n}_a\}$ and $\{\mathtt{n}_b\}$, the lambda expressions $(\lambda_{\mathtt{n}_a}\ (\mathtt{a})\ \mathtt{a})$ and $(\lambda_{\mathtt{n}_b}\ (\mathtt{b})\ \mathtt{b})$ have

the different types $i \xrightarrow{\{\mathtt{n}_a\}} i$ and $i \xrightarrow{\{\mathtt{n}_b\}} i$. Therefore `f` cannot keep the same type at the different call contexts $\mathtt{l}_a$ and $\mathtt{l}_b$. Thus the program is rejected by type checking.

Subeffecting has been used to improve the flexibility of effect systems [Talpin92-1, Dornic91, Tang92]. It allows expressions to admit larger effects when a type mismatch occurs. The subeffecting approach forces a function to have identical type in different call contexts by merging their effects together, thus limiting the accuracy of effect systems.

Subeffecting is introduced by the (*does*) rule that allows an expression to admit a larger effect c instead of c'. This rule can be used whenever a type mismatch occurs in the application rule (*app*) due to its latent effect.

$$(does): \frac{\mathcal{E} \vdash \mathtt{e} : t, c' \quad c' \subseteq c}{\mathcal{E} \vdash \mathtt{e} : t, c}$$

For the same example `demo`, by using the subeffecting technique, the lambda expressions $(\lambda_{\mathtt{n}_a}\ (\mathtt{a})\ \mathtt{a})$ and $(\lambda_{\mathtt{n}_b}\ (\mathtt{b})\ \mathtt{b})$ admit a unique type $i \xrightarrow{\{\mathtt{n}_a,\mathtt{n}_b\}} i$. Therefore `f` can keep the same type $(i \xrightarrow{\{\mathtt{n}_a,\mathtt{n}_b\}} i) \xrightarrow{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}} i$ at the different call sites $\mathtt{l}_a$ and $\mathtt{l}_b$, thus this program can be type checked. Note that subeffecting forces the latent effects of the lambda expressions $(\lambda_{\mathtt{n}_a}\ (\mathtt{a})\ \mathtt{a})$ and $(\lambda_{\mathtt{n}_b}\ (\mathtt{b})\ \mathtt{b})$ to be merged together, thus limiting the accuracy of effect systems.

4.3 Subeffecting-Based Reconstruction

We present an algorithm based on subeffecting for reconstructing types and control-flow effects of expressions. To stay as close as possible to classical unification-based algorithms, the basic idea is that latent effects of functions are always represented by effect variables. Thus the type unification can be applied on both type and effect variables. Beside reconstructing types and control-flow effects of expressions, the algorithm also computes a set of subsumption constraints between effect variables and control-flow effects of the form $\{\zeta_i \supseteq c_i \mid i = 1..s\}$. Thus the reconstruction of types and control-flow effects of expressions is viewed as a constraint satisfaction problem. For an expression that admits a type and control-flow effect in the static semantics, its corresponding effect constraint set must have at least one solution. This solution satisfies the criteria of the maximality of the type with respect to substitution on type variables, and minimality of the effect with respect to the subsumption on effect variables.

$$\kappa \quad \in \text{Constraint} \quad = \mathcal{P}(\text{EfVar} * \text{Control})$$

4.3.1 Unification

The unification algorithm $\mathcal{U}$ [Robinson65] solves the equations on types and effect variables built by the reconstruction algorithm. It returns a substitution θ as the most general unifier of two terms, or fails. Note that the reconstruction algorithm only unifies effect variables. Substitutions θ are defined as maps from type or effect variables to their corresponding values. We note Id is the identity substitution.

$$\theta \quad \in \text{Subst} \quad = (\text{TyVar} \mapsto \text{Type}) + (\text{EfVar} \mapsto \text{Control})$$

$\mathcal{U}(t,t') = \mathit{case}\ (t,t')\ \mathit{of}$

$$\begin{array}{ll}
(\mathit{int},\mathit{int}) & \Rightarrow Id \\
(\alpha,\alpha') & \Rightarrow [\alpha \mapsto \alpha'] \\
(\alpha,t)|(t,\alpha) & \Rightarrow \mathit{if}\ \alpha \in \mathit{fv}(t)\ \mathit{then}\ \text{fail}\ \mathit{else}\ [\alpha \mapsto t] \\
(t_0 \xrightarrow{\zeta_0} t'_0, t_1 \xrightarrow{\zeta_1} t'_1) & \Rightarrow \mathit{let}\ \theta = [\zeta_0 \mapsto \zeta_1]\ \mathit{and}\ \theta' = \mathcal{U}(\theta t_0, \theta t_1) \\
 & \quad \mathit{in}\ \ \mathcal{U}(\theta'\theta t'_0, \theta'\theta t'_1)\theta'\theta \\
(_,_) & \Rightarrow \text{fail}
\end{array}$$

where for any substitutions θ and θ', θt is the application of the substitution θ to t and $\theta\theta'$ is the composition of the substitutions with the property that $\theta\theta'(\vartheta) = \theta(\theta'(\vartheta))$.

Lemma 4.1 (Correctness of $\mathcal{U}$) *Let t and t' be two type terms in the domain of $\mathcal{U}$. If $\theta = \mathcal{U}(t,t')$, then*

- *$\mathcal{U}$ is sound : $\theta t = \theta t'$,*
- *$\mathcal{U}$ is complete : If $\theta' t = \theta' t'$, there exists a substitution θ'' such that $\theta' = \theta''\theta$.*

Proof *$\mathcal{U}$ unifies terms over a free algebra, and is thus complete following [Robinson65].*

4.3.2 Algorithm $\mathcal{R}$

Given an expression `e` and its type environment $\mathcal{E}$, the reconstruction algorithm $\mathcal{R}$ computes a substitution θ, its type t, control-flow effect c and a constraint set κ. We note :

$$\mathcal{R}(\mathcal{E}, \mathtt{e}) = \langle \theta, t, c, \kappa \rangle$$

The effect constraint set is built during the reconstruction of `lambda` expressions where a latent effect is introduced into the function type.

$\mathcal{R}(\mathcal{E}, \mathtt{x}) \Rightarrow$
  if $\mathtt{x} \in \mathit{Dom}(\mathcal{E})$
  then $\langle Id, \mathcal{E}(\mathtt{x}), \emptyset, \emptyset \rangle$
  else fail

$\mathcal{R}(\mathcal{E}, (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e})) \Rightarrow$
  let α, ζ *new*
    $\langle \theta, t, c, \kappa \rangle = \mathcal{R}(\mathcal{E}[\mathtt{x} \mapsto \alpha], \mathtt{e})$
  in $\langle \theta, \theta\alpha \xrightarrow{\zeta} t, \emptyset, \kappa \cup \{\zeta \supseteq c \cup \{\mathtt{n}\}\} \rangle$

$\mathcal{R}(\mathcal{E}, (\mathtt{rec}_{\mathtt{n}}\ (\mathtt{f}\ \mathtt{x})\ \mathtt{e})) \Rightarrow$
  let α', α, ζ *new*
    $\mathcal{E}' = \mathcal{E}[\mathtt{f} \mapsto \alpha' \xrightarrow{\zeta} \alpha][\mathtt{x} \mapsto \alpha']$

$$\begin{array}{l} \quad \langle \theta, t, c, \kappa \rangle = \mathcal{R}(\mathcal{E}', \mathtt{e}) \\ \quad \theta' = \mathcal{U}(\theta\alpha, t) \\ \quad in\ \langle \theta'\theta, \theta'\theta(\alpha' \xrightarrow{\zeta} \alpha), \emptyset, \theta'(\kappa \cup \{\theta\zeta \supseteq \{\mathtt{n}\} \cup c\}) \rangle \end{array}$$

$$\begin{array}{l} \mathcal{R}(\mathcal{E}, (\mathtt{e}\ \mathtt{e}')) \Rightarrow \\ \quad let\ \langle \theta, t, c, \kappa \rangle = \mathcal{R}(\mathcal{E}, \mathtt{e}) \\ \quad\quad \langle \theta', t', c', \kappa' \rangle = \mathcal{R}(\theta\mathcal{E}, \mathtt{e}') \\ \quad\quad \alpha, \zeta\ new \\ \quad\quad \theta'' = \mathcal{U}(\theta' t, t' \xrightarrow{\zeta} \alpha) \\ \quad in\ \langle \theta''\theta'\theta, \theta''\alpha, \theta''(\theta' c \cup c' \cup \zeta), \theta''(\theta'\kappa \cup \kappa') \rangle \end{array}$$

Note that by the unification of two effect variables ζ and ζ', the inequalities $\{\zeta \supseteq c, \zeta' \supseteq c'\}$ in the constraint set are replaced by $\{\zeta \supseteq c, \zeta \supseteq c'\}$, which is equivalent to $\{\zeta \supseteq c \cup c'\}$.

4.3.3 Constraint Satisfaction

An expression $\mathtt{e}$ with its type environment $\mathcal{E}$ is type and effect safe if and only if $\mathcal{R}(\mathcal{E}, \mathtt{e})$ does not fail and the returned constraint set κ admits at least one solution. The constraint set is of the *normal form* $\{\zeta_i \supseteq c_i \mid i = 1..s\}$, which guarantees the existence of solutions. We are interested at the minimal one.

The substitutions satisfying κ are called *effect models*.

Definition 4.1 (Effect Model) *A substitution μ is an* effect model *of a constraint set κ, noted as $\mu \models \kappa$, if and only if $\forall \zeta \supseteq c \in \kappa$, $\mu\zeta \supseteq \mu c$.*

Theorem 4.1 (Satisfaction) *Every normal form constraint set $\kappa = \{\zeta_i \supseteq c_i \mid i = 1..s\}$ admits at least one effect model.*

Proof *$\{\zeta_i \mapsto c_i' \mid i = 1 \ldots n\}$ is an effect model of κ, where $c_i' = \cup_{i=1}^{n} c_i \setminus \cup_{i=1}^{n} \zeta_i$.*

A constraint set may admit more than one effect model, among which we are interested in the minimal one. We define a function *Min* to characterize the minimal effect model of a constraint set κ. Note that the solution is independent of the order of inequalities in κ.

$$\begin{array}{rl} Min(\emptyset) & = \mathrm{Id} \\ Min(\{\zeta \supseteq c\} \cup \kappa') & = let\ \mu = Min(\kappa')\ in\ \{\zeta \mapsto \mu c \setminus \zeta\}\mu \end{array}$$

The constraint set of the reconstruction algorithm always admits a unique minimal model with respect to the subsumption relation $\supseteq$ on effects.

Theorem 4.2 (Minimality) *Any constraint set admits a unique minimal effect model.*

Proof *By induction on the constraint set.*

4.3.4 Correctness

The reconstruction algorithm terminates and is sound and complete with respect to the static semantics.

Theorem 4.3 (Termination) *For all inputs* $(\mathcal{E}, \mathtt{e})$*, the algorithm* $\mathcal{R}$ *either fails or terminates.*

Proof *Since the reconstruction algorithm* $\mathcal{R}$ *is defined on the structure of expressions of finite height, it terminates or fails.*

The soundness theorem states that the application of any effect model of the reconstructed constraint set to the reconstructed type and effect satisfies the static semantics.

Theorem 4.4 (Soundness) *Given an expression* $\mathtt{e}$ *and its type environment* $\mathcal{E}$*, if* $\mathcal{R}(\mathcal{E}, \mathtt{e}) = \langle \theta, t, c, \kappa \rangle$*, then, for any effect model* μ *of* κ*, one has:*

$$\mu\theta\mathcal{E} \vdash \mathtt{e} : \mu t, \mu c$$

The completeness theorem states that the reconstructed type t and control-flow effect c are maximal with respect to any type t_1 and control-flow effect c_1 derivable by the static semantics, for some substitution μ that satisfies the computed constraint set κ.

Theorem 4.5 (Completeness) *If* $\theta_1\mathcal{E} \vdash \mathtt{e} : t_1, c_1$*, then* $\mathcal{R}(\mathcal{E}, \mathtt{e}) = \langle \theta, t, c, \kappa \rangle$ *and there exists an effect model* μ *of* κ *such that:*

$$\theta_1\mathcal{E} = \mu\theta\mathcal{E} \quad \textit{and} \quad t_1 = \mu t \quad \textit{and} \quad c_1 \supseteq \mu c$$

4.3.5 Example

We use the same example `demo` to show the reconstruction process by solving the effect constraint set. Here we give the following table to show the type of the variables $\mathtt{f}, \mathtt{g}$ and the lambda expressions $\mathtt{n}_f, \mathtt{n}_a, \mathtt{n}_b, \mathtt{n}_g$.

$$\begin{array}{ll}
\mathtt{f}: & (i \xrightarrow{\zeta_1} i) \xrightarrow{\zeta_2} i \\
\mathtt{g}: & i \xrightarrow{\zeta_1} i \\
\mathtt{n}_f: & ((i \xrightarrow{\zeta_1} i) \xrightarrow{\zeta_2} i) \xrightarrow{\zeta_3} i \\
\mathtt{n}_a: & i \xrightarrow{\zeta_1} i \\
\mathtt{n}_b: & i \xrightarrow{\zeta_1} i \\
\mathtt{n}_g: & (i \xrightarrow{\zeta_1} i) \xrightarrow{\zeta_2} i
\end{array}$$

The constraint set is of the form:

$$\{\zeta_1 \supseteq \{\mathtt{n}_a, \mathtt{n}_b\}, \quad \zeta_2 \supseteq \{\mathtt{n}_g\} \cup \zeta_1, \quad \zeta_3 \supseteq \{\mathtt{n}_f\} \cup \zeta_2\}$$

The minimal solution of this constraint set is as below :

$$\zeta_1 \mapsto \{\mathtt{n}_a, \mathtt{n}_b\}, \quad \zeta_2 \mapsto \{\mathtt{n}_g, \mathtt{n}_a, \mathtt{n}_b\}, \quad \zeta_3 \mapsto \{\mathtt{n}_f, \mathtt{n}_g, \mathtt{n}_a, \mathtt{n}_b\}$$

4.4 System Extensions

The subeffecting effect system can be straightforwardly extended to imperative constructs and let-bound expressions in a way as presented in [Talpin92-1]. A reference type $ref(t)$ needs to be introduced to represent updatable memory locations containing values of the type t.

The parametric polymorphism introduced in ML type systems [Milner78, Tofte87] can be extended to effect systems [Talpin92-1]. The type of $\mathtt{e}'$ in (`let` (`x` $\mathtt{e}'$) `e`) can be generalized on both type and effect variables, if $\mathtt{e}'$ is side-effect free. Therefore its effect variables can be instantiated to different effects in different call contexts, thus increasing the accuracy of effect systems. [Tofte87] introduces an *expansiveness* function to detect if expressions have side-effects or not. A non-expansive expression is syntactically guaranteed to be side-effect free. By using side-effect analysis [Talpin92-1], especially by introducing an *observation* criterion [Talpin92-2], type generalization in let-expressions can be performed in a more precise way. One can use syntactic substitution to avoid the complication of introducing sophisticated type schemes. We write $\mathtt{e}'[\mathtt{e}/\mathtt{x}]$ for the textual substitution of `e` for `x` in $\mathtt{e}'$. The syntactic substitution $\mathtt{e}'[\mathtt{e}/\mathtt{x}]$ reflects the fact that different types of `e` can be used for each occurrence of `x` in $\mathtt{e}'$. In the static semantics, the type generalization of let expression is specified by the following (*let*) rule :

$$(let): \quad \frac{\begin{array}{l}\neg expansive[\![\mathtt{e}]\!] \\ \mathcal{E} \vdash \mathtt{e} : t, \emptyset \\ \mathcal{E} \vdash \mathtt{e}'[\mathtt{e}/\mathtt{x}] : t', c'\end{array}}{\mathcal{E} \vdash (\mathtt{let}\ (\mathtt{x}\ \mathtt{e})\ \mathtt{e}') : t', c'}$$

In the reconstruction algorithm, in order to avoid recomputing the type of expressions bound in *let* constructs, we use algebraic type schemes to generalize their types and associated constraints. Here *algebraic type schemes*, noted $\forall v_{1..n}.(t, \kappa)$, are composed of a type t and a set of inequalities κ universally quantified over type and effect variables $v_{1..n}$. Algebraic type schemes are used to implement the textual substitution specified in the *let* rule in the statics semantics. The type and constraint set associated with `e` only depend on the free variables of `e` and, thereby, on the type environment $\mathcal{E}$. An algebraic type scheme *caches* the effect constraint that would have to be recomputed each time `e` appeared in the substituted body. Constrained type environments $\mathcal{E}$ map value identifiers to algebraic type schemes.

$\forall v_{1..n}.(t, \kappa)$	$\in$ TyScheme		*type scheme*
$\mathcal{E}$	$\in$ TEnv	$=$ Id $\mapsto$ TyScheme	*type environment*

The reconstruction algorithm only needs to be modified for the case of identifiers and let-bound expressions.

$\mathcal{R}(\mathcal{E}, \mathtt{x}) \Rightarrow$
if $\mathtt{x} \mapsto \forall v_{1..n}.(t, \kappa) \in \mathcal{E}$
then *let* $\{v'_{1..n}\}$ *new*
$\theta = \{v_i \mapsto v'_i \mid i = 1 \ldots n\}$
in $\langle \theta, \theta t, \emptyset, \theta\kappa \rangle$
else fail

$\mathcal{R}(\mathcal{E}, (\texttt{let (x e) e}')) \Rightarrow$
$\quad let\ \langle \theta, t, c, \kappa \rangle = \mathcal{R}(\mathcal{E}, \texttt{e})$
$\quad in\ \ let\ v_{1..n} = (fv(t) \cup fv(\kappa)) \setminus fv(\mathcal{E})$
$\qquad \mathcal{E}' = \theta\mathcal{E}_{\texttt{x}} \cup \{\texttt{x} \mapsto \forall v_{1..n}.(t, \kappa)\}$
$\qquad \langle \theta', t', c', \kappa' \rangle = \mathcal{R}(\mathcal{E}', \texttt{e}')$
$\qquad in\ \ \langle \theta'\theta, t', c', \kappa' \rangle$

The example program `demo` can be equivalently written in the let-binding form.

$$\begin{array}{l}\texttt{let-demo=}\ (\texttt{let}\ \ (\texttt{f}\ (\lambda_{\texttt{n}_g}\ (\texttt{g})\ (\texttt{g}\ 1)_{\texttt{l}_g})) \\ \qquad\qquad\qquad (+\ (\texttt{f}\ (\lambda_{\texttt{n}_a}\ (\texttt{a})\ \texttt{a}))_{\texttt{l}_a} \\ \qquad\qquad\qquad\quad (\texttt{f}\ (\lambda_{\texttt{n}_b}\ (\texttt{b})\ \texttt{b}))_{\texttt{l}_b}))\end{array}$$

The type of the let-binding expression $(\lambda_{\texttt{n}_g}\ (\texttt{g})\ (\texttt{g}\ 1))$ can be generalized to $\forall\zeta_1\zeta_2.((i \xrightarrow{\zeta_1} i) \xrightarrow{\zeta_2} i, \emptyset)$. When it is called (via `f`) at the call site $\texttt{l}_a$ and $\texttt{l}_b$, it can be instantiated to $(i \xrightarrow{\zeta_1'} i) \xrightarrow{\zeta_2'} i$ and $(i \xrightarrow{\zeta_1''} i) \xrightarrow{\zeta_2''} i$ respectively and the constraint set is as below:

$$\{\zeta_1' \supseteq \{\texttt{n}_a\},\ \ \zeta_1'' \supseteq \{\texttt{n}_b\},\ \ \zeta_2' \supseteq \{\texttt{n}_g\} \cup \zeta_1',\ \ \zeta_2'' \supseteq \{\texttt{n}_g\} \cup \zeta_1'',\ \ \zeta_3 \supseteq \{\texttt{n}_f\} \cup \zeta_2' \cup \zeta_2''\}$$

By solving the constraint set, we get the minimal solution of this constraint set :

$$\zeta_1' \mapsto \{\texttt{n}_a\},\ \ \zeta_1'' \mapsto \{\texttt{n}_b\},\ \ \zeta_2' \mapsto \{\texttt{n}_g, \texttt{n}_a\},\ \ \zeta_2'' \mapsto \{\texttt{n}_g, \texttt{n}_b\},\ \ \zeta_3 \supseteq \{\texttt{n}_f, \texttt{n}_g, \texttt{n}_a, \texttt{n}_b\}$$

Therefore the control-flow information at the call site $\texttt{l}_a$ or $\texttt{l}_b$ is $\{\texttt{n}_g, \texttt{n}_a\}$ or $\{\texttt{n}_g, \texttt{n}_b\}$ respectively. With the previously defined subeffecting effect system, the control-flow information of $(\texttt{f}\ (\lambda_{\texttt{n}_a}\ (\texttt{a})\ \texttt{a}))$ and $(\texttt{f}\ (\lambda_{\texttt{n}_b}\ (\texttt{b})\ \texttt{b}))$ are both $\{\texttt{n}_g, \texttt{n}_a, \texttt{n}_b\}$, even if only $\{\texttt{n}_g, \texttt{n}_a\}$ or $\{\texttt{n}_g, \texttt{n}_b\}$ are called during their evaluations, thus collapsing different call contexts together.

Here we give the following table to compare the control-flow information at the call sites $\texttt{l}_a$ and $\texttt{l}_b$ collected by effect systems with and without the introduction of the generic types. Note that the introduction of generic types can distinguish different call contexts ($\texttt{n}_a$ and $\texttt{n}_b$), thus performing more precise control-flow analysis.

	Dynamic Semantics	Effect Systems	Generic Types
$\texttt{l}_a$	$\{\texttt{n}_g, \texttt{n}_a\}$	$\{\texttt{n}_g, \texttt{n}_a, \texttt{n}_b\}$	$\{\texttt{n}_g, \texttt{n}_a\}$
$\texttt{l}_b$	$\{\texttt{n}_g, \texttt{n}_b\}$	$\{\texttt{n}_g, \texttt{n}_a, \texttt{n}_b\}$	$\{\texttt{n}_g, \texttt{n}_b\}$

4.5 Conclusion

We introduced subeffecting to improve the flexibility of effect systems. The subeffecting approach allows expressions to admit larger effects when a type mismatch occurs. We presented the subeffecting rule in the static semantics, discussed the reconstruction algorithms $\mathcal{R}$ and proved it sound and complete w.r.t. the static semantics. Finally, we described how to extend generic polymorphism in effect systems. This polymorphic control-flow effect system has been implemented in a Mini-FX compiler [Grundman92] where Mini-FX is a variant of the Scheme programming language [MIT90].

Chapter 5

Subtyping Effect Systems

Subtyping improves both flexibility and accuracy of effect systems.

5.1 Introduction

Instead of subeffecting, we introduce subtyping in effect systems. Subtyping allows a function to have different types in different call contexts, as long as they obey certain subtype relation. The subtype relation is induced by a subsumption rule on effects. This *subtyping effect system* avoids effect information to be merged together when forcing two types to be identical, thus collecting more precise effect information than the previous effect system based on subeffecting. We introduce a reconstruction algorithm that for each expression computes its type and control-flow effect based on subtyping. The reconstruction algorithm is sound and complete w.r.t. the static semantics.

We introduce the subtyping rule (Section 5.2), present the corresponding reconstruction algorithm and prove it sound and complete w.r.t. the static semantics (Section 5.3). Finally we describe related work (Section 5.4) before concluding (Section 5.5). All proofs are presented in Appendix 3.

5.2 The Subtyping Rule

The subtype relation is defined between types of the same structure. The type structure is defined by the classical types. A classical type τ can either be int, a type variable α, or a function type $\tau' \to \tau$.

$$\tau \in \text{CType} \quad = int \mid \alpha \mid \tau' \to \tau \quad \textit{classical type}$$

An effect, here specified by a set of function names, can be conservatively approximated by one of its supersets. The subeffect relation is thus the usual set inclusion relation.

If two function types have the same structure, the subtype relation $\leq$ is defined via the inclusion relation between their latent effects. To properly define this notion,

we introduce the *Erase* function which transforms types to classical types by deleting latent effects.

$$\begin{array}{ll} Erase(int) & = int \\ Erase(\alpha) & = \alpha \\ Erase(t' \xrightarrow{c} t) & = Erase(t') \rightarrow Erase(t) \end{array}$$

The *type structure* of t is $Erase(t)$. Two types t and t' have the same structure if and only if $Erase(t) = Erase(t')$.

The subtype relation $t \leq t'$ is defined whenever t and t' have the same structure. Note that the subtype relation between function types is contravariant.

Definition 5.1 (Subtype)

$$\begin{array}{rcl} int \leq int & & \\ \alpha \leq \alpha & & \\ t'_0 \xrightarrow{c_0} t_0 \ \leq \ t'_1 \xrightarrow{c_1} t_1 & \Leftrightarrow & t'_1 \leq t'_0 \ \wedge \ t_0 \leq t_1 \ \wedge c_1 \supseteq c_0 \end{array}$$

The function *Eff* generates the set of effect inequalities corresponding to a given type inequality. An effect inequality is a pair (c_i, c'_i) written $c_i \supseteq c'_i$.

$$\begin{array}{ll} Eff(int \leq int) & = \emptyset \\ Eff(\alpha \leq \alpha) & = \emptyset \\ Eff(t'_0 \xrightarrow{c_0} t_0 \leq t'_1 \xrightarrow{c_1} t_1) & = \{c_1 \supseteq c_0\} \cup Eff(t'_1 \leq t'_0) \cup Eff(t_0 \leq t_1) \end{array}$$

Subtyping is introduced by the following (*sub*) rule that allows a larger type t to be used in lieu of t'. This increases the flexibility of the static semantics by allowing a function to admit different types, as long as a certain subtype relation is satisfied. It avoids merging effect information together when forcing two types to be identical, thus collecting more precise effect information than the subeffecting technique.

$$(sub): \frac{\mathcal{E} \vdash \mathtt{e} : t' \quad t' \leq t}{\mathcal{E} \vdash \mathtt{e} : t}$$

We use the same example `demo` to show how subtyping improves the accuracy of effect systems compared to subeffecting. When the function `f` (bound to $\mathtt{n}_g$) is called at $\mathtt{l}_a$ and $\mathtt{l}_b$, it can admit the supertypes $(i \xrightarrow{\{\mathtt{n}_a\}} i) \xrightarrow{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}} i$ and $(i \xrightarrow{\{\mathtt{n}_b\}} i) \xrightarrow{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}} i$ respectively instead of having the unique type $(i \xrightarrow{\{\mathtt{n}_a,\mathtt{n}_b\}} i) \xrightarrow{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}} i$. The following table shows the difference between subeffecting and subtyping. We give the types of `f` at these three occurrences (t_f, t'_f and t''_f), and the types of the arguments $\mathtt{n}_a$ and $\mathtt{n}_b$ ($t_{\mathtt{n}_a}$ and $t_{\mathtt{n}_b}$)

	Subeffecting	Subtyping
t_f	$(i \overset{\{\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i) \overset{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i$	$(i \overset{\{\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i) \overset{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i$
t'_f	$(i \overset{\{\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i) \overset{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i$	$(i \overset{\{\mathtt{n}_a\}}{\rightarrow} i) \overset{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i$
$t_{\mathtt{n}_a}$	$i \overset{\{\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i$	$i \overset{\{\mathtt{n}_a\}}{\rightarrow} i$
t''_f	$(i \overset{\{\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i) \overset{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i$	$(i \overset{\{\mathtt{n}_b\}}{\rightarrow} i) \overset{\{\mathtt{n}_g,\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i$
$t_{\mathtt{n}_a}$	$i \overset{\{\mathtt{n}_a,\mathtt{n}_b\}}{\rightarrow} i$	$i \overset{\{\mathtt{n}_b\}}{\rightarrow} i$
	$t_f = t'_f$ $t_f = t''_f$	$t_f \leq t'_f$ $t_f \leq t''_f$

Notice that, when using subeffecting, all occurrences of f are forced to have the same type, while, when using subtyping, they only have to obey a subtype relation, leading to more precise local control-flow information.

5.3 Subtyping-Based Reconstruction

The reconstruction of types and effects based on subtyping operates on the expressions that are already typed by classical type systems. For each expression `e` annotated with classical types, the reconstruction algorithm $\mathcal{S}$ computes its type, its effect and a set of type inequalities based on subtyping. Since the subtype relation in our system is defined by the subsumption relation on effects and the structures of the types are known, type inequalities amount to sets of effect inequalities. Therefore solving type inequalities is reduced to solving the corresponding effect inequalities. Thus the subtyping-based reconstruction can be also viewed as an effect constraint satisfaction problem. For every expression that has a type and a control-flow effect in the static semantics, its effect constraint set must have at least one solution, which satisfies the set of type inequalities.

For any expression `e`, the classical type reconstruction algorithm computes its principal type τ w.r.t. the types t defined by the static semantics (modulo *Erase*). In other words, for any expression, if τ is its type computed by the classical type reconstruction algorithm and t is derivable from the static semantics, then there exists a type substitution θ, such that :

$$Erase(t) = \theta\tau$$

5.3.1 Algorithm $\mathcal{S}$

Given a type environment $\mathcal{E}$ and an expression `e` assumed priorly decorated with its classical type (we use a straightforward expression annotation mechanism to express this information in the algorithm), the reconstruction algorithm $\mathcal{S}$ computes a type t, an effect c and an effect constraint set κ. We note :

$$\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t, c, \kappa \rangle$$

The effect constraint set is partly built by application of *Eff* to type inequalities and partly during the reconstruction of `lambda` and `rec` expressions:

$$
\begin{array}{l}
\mathcal{S}(\mathcal{E}, \mathtt{x}) \Rightarrow \\
\quad \mathit{let}\ \ t' = \mathcal{E}(\mathtt{x}) \\
\qquad\ \ t = \mathit{New}(\mathit{Erase}(t')) \\
\quad \mathit{in}\ \ \langle t, \emptyset, \mathit{Eff}(t' \leq t)\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}(\mathcal{E}, (\lambda_{\mathtt{n}}\ (\mathtt{x} : \tau)\ \mathtt{e})) \Rightarrow \\
\quad \mathit{let}\ \ t' = \mathit{New}(\tau) \\
\qquad\ \ \zeta\ \mathit{new} \\
\qquad\ \ \langle t, c, \kappa\rangle = \mathcal{S}(\mathcal{E}[\mathtt{x} \mapsto t'], \mathtt{e}) \\
\quad \mathit{in}\ \ \langle t' \xrightarrow{\zeta} t, \emptyset, \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}(\mathcal{E}, (\mathtt{rec}_{\mathtt{n}}\ (\mathtt{f} : \tau' \to \tau\ \ \mathtt{x} : \tau')\ \mathtt{e})) \Rightarrow \\
\quad \mathit{let}\ \ t' \xrightarrow{\zeta} t = \mathit{New}(\tau' \to \tau) \\
\qquad\ \ \langle t'', c, \kappa\rangle = \mathcal{S}(\mathcal{E}[\mathtt{f} \mapsto t' \xrightarrow{\zeta} t][\mathtt{x} \mapsto t'], \mathtt{e}) \\
\quad \mathit{in}\ \ \langle t' \xrightarrow{\zeta} t, \emptyset, \kappa \cup \mathit{Eff}(t'' \leq t) \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}(\mathcal{E}, (\mathtt{e}\ \mathtt{e}')) \Rightarrow \\
\quad \mathit{let}\ \langle t'' \xrightarrow{c''} t, c, \kappa\rangle = \mathcal{S}(\mathcal{E}, \mathtt{e}) \\
\qquad\ \ \langle t', c', \kappa'\rangle = \mathcal{S}(\mathcal{E}, \mathtt{e}') \\
\quad \mathit{in}\ \ \langle t, c \cup c' \cup c'', \kappa \cup \kappa' \cup \mathit{Eff}(t' \leq t'')\rangle
\end{array}
$$

where the function New transforms a classical type τ to a type t by adding fresh latent effect variables to τ. Its proper definition is:

$$
\begin{array}{lll}
\mathit{New}(\mathit{int}) & = \mathit{int} & \\
\mathit{New}(\alpha) & = \alpha & \\
\mathit{New}(\tau' \to \tau) & = \mathit{New}(\tau') \xrightarrow{\zeta} \mathit{New}(\tau) & \textit{for fresh } \zeta
\end{array}
$$

Note that subeffecting can be easily related to subtyping by noticing that its related reconstruction algorithm $\mathcal{R}$ is similar to $\mathcal{S}$, except that $\leq$ is replaced by the more restrictive $=$, implemented by unification.

5.3.2 Properties of $\mathcal{S}$

We can easily prove by induction that the reconstruction algorithm $\mathcal{S}$ has the following properties :

Lemma 5.1 (Properties of $\mathcal{S}$) *For any $\mathcal{E}$, $\mathtt{e}$, if $\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t, c, \kappa\rangle$, then :*

- *t only includes fresh effect variables.*
- *All environment extensions within $\mathcal{S}$ refer to types with only fresh effect variables.*

The previous lemma implies that the constraint set computed by the reconstruction algorithm $\mathcal{S}$ has the following normal form property:

Lemma 5.2 (Normal Effect Constraints) *If $\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t, c, \kappa\rangle$, then κ has the following normal form:*

$$\{\zeta_i \supseteq c_i \mid i = 1..s\}$$

As stated in Chapter 4, effect constraint sets in this normal form guarantee the existence of unique minimal effect models. The following lemma shows how to satisfy a type inequality by solving its corresponding effect constraint.

Lemma 5.3 (Solution of Type Inequality) *If t and t' have the same structure, then*

$$\mu \models Eff(t' \leq t) \Leftrightarrow \mu t' \leq \mu t$$

Proof *By induction of the structure of types.*

5.3.3 Correctness

Since the reconstruction algorithm $\mathcal{S}$ is defined by induction on the structure of expressions, which are of finite height, it always terminates. It is sound and complete with respect to the static semantics.

The soundness theorem states that the application of any effect model of the reconstructed type constraint set to the reconstructed type and effect satisfies the static semantics.

Theorem 5.1 (Soundness) *Given an expression* $\mathtt{e}$ *and its type environment $\mathcal{E}$, if $\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t, c, \kappa \rangle$, then for any effect model μ of κ, one has:*

$$\mu\mathcal{E} \vdash \mathtt{e} : \mu t, \mu c$$

The completeness theorem states that the reconstructed type t and the control-flow effect c are maximal with respect to any type t_1 and control-flow effect c_1 derivable from the static semantics, modulo some substitution μ that satisfies the computed constraint set κ.

Theorem 5.2 (Completeness) *If $\theta_1\mathcal{E} \vdash \mathtt{e} : t_1, c_1$, then $\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t, c, \kappa \rangle$ and there exists an effect model μ of κ, such that:*

$$\theta_1\mathcal{E} = \mu\mathcal{E} \quad and \quad \mu t \leq t_1 \quad and \quad c_1 \supseteq \mu c$$

5.3.4 Example

We use the same example `demo` to show how the reconstruction algorithm $\mathcal{S}$ works for expressions already typed with classical types. After applying the classical type reconstruction algorithm, the program `demo` is annotated by classical types like below :

$$\begin{array}{l}\mathtt{demo\text{-}annotated}=((\lambda_{\mathtt{n}_f}\ \ (\mathtt{f} : (i \to i) \to i) \\ \qquad\qquad\qquad\qquad (+\ (\mathtt{f}\ (\lambda_{\mathtt{n}_a}\ (\mathtt{a} : i \to i)\ \mathtt{a}))_{\mathtt{l}_a} \\ \qquad\qquad\qquad\qquad\quad (\mathtt{f}\ (\lambda_{\mathtt{n}_b}\ (\mathtt{b} : i \to i)\ \mathtt{b}))_{\mathtt{l}_b})\ ) \\ \qquad\qquad\qquad (\lambda_{\mathtt{n}_g}\ (\mathtt{g} : i \to i)\ (\mathtt{g}\ 1)_{\mathtt{l}_g})_{\mathtt{l}_f})\end{array}$$

We apply the algorithm $\mathcal{S}$ to `demo-annotated`. Here we give the following table to show the type of the variable $\mathtt{f}, \mathtt{g}$ and the lambda expression $\mathtt{n}_a, \mathtt{n}_b, \mathtt{n}_g, \mathtt{n}_f$. Note that the variable `f` has different types in its three occurrences $\mathtt{l}_f$, $\mathtt{l}_a$ and $\mathtt{l}_b$.

$$
\begin{array}{ll}
\mathtt{f}(\mathtt{l}_f): & (i \xrightarrow{\zeta_1} i) \xrightarrow{\zeta_2} i \\
\mathtt{f}(\mathtt{l}_a): & (i \xrightarrow{\zeta_1'} i) \xrightarrow{\zeta_2'} i \\
\mathtt{f}(\mathtt{l}_b): & (i \xrightarrow{\zeta_1''} i) \xrightarrow{\zeta_2''} i \\
\mathtt{g}: & i \xrightarrow{\zeta_g'} i \\
\mathtt{n}_a: & i \xrightarrow{\zeta_a} i \\
\mathtt{n}_b: & i \xrightarrow{\zeta_b} i \\
\mathtt{n}_g: & (i \xrightarrow{\zeta_g'} i) \xrightarrow{\zeta_g} i \\
\mathtt{n}_f: & ((i \xrightarrow{\zeta_1} i) \xrightarrow{\zeta_2} i) \xrightarrow{\zeta_3} i
\end{array}
$$

The constraint set is the union of following inequalities:

$$
\begin{array}{l}
\mathit{Eff}((i \xrightarrow{\zeta_1} i) \xrightarrow{\zeta_2} i \le (i \xrightarrow{\zeta_1'} i) \xrightarrow{\zeta_2'} i), \\
\zeta_a \supseteq \{\mathtt{n}_a\}, \\
\mathit{Eff}(i \xrightarrow{\zeta_a} i \le i \xrightarrow{\zeta_1'} i), \\
\mathit{Eff}((i \xrightarrow{\zeta_1} i) \xrightarrow{\zeta_2} i \le (i \xrightarrow{\zeta_1''} i) \xrightarrow{\zeta_2''} i), \\
\zeta_b \supseteq \{\mathtt{n}_b\}, \\
\mathit{Eff}(i \xrightarrow{\zeta_b} i \le i \xrightarrow{\zeta_1''} i), \\
\zeta_3 \supseteq \{\mathtt{n}_f\} \cup \zeta_2' \cup \zeta_2'', \\
\zeta_g \supseteq \{\mathtt{n}_g\} \cup \zeta_g', \\
\mathit{Eff}((i \xrightarrow{\zeta_g'} i) \xrightarrow{\zeta_g} i \le (i \xrightarrow{\zeta_1} i) \xrightarrow{\zeta_2} i)
\end{array}
$$

which is equivalent to :

$$
\begin{array}{l}
\{\zeta_2' \supseteq \zeta_2, \quad \zeta_1 \supseteq \zeta_1', \\
\zeta_a \supseteq \{\mathtt{n}_a\}, \\
\zeta_1' \supseteq \zeta_a, \\
\zeta_2'' \supseteq \zeta_2, \quad \zeta_1 \supseteq \zeta_1'', \\
\zeta_b \supseteq \{\mathtt{n}_b\}, \\
\zeta_1'' \supseteq \zeta_b, \\
\zeta_3 \supseteq \{\mathtt{n}_f\} \cup \zeta_2' \cup \zeta_2'', \\
\zeta_g \supseteq \{\mathtt{n}_g\} \cup \zeta_g', \\
\zeta_2 \supseteq \zeta_g, \quad \zeta_g' \supseteq \zeta_1\}
\end{array}
$$

The minimal solution of this constraint set is as below :

$$\begin{array}{l}
\zeta_a \mapsto \{\mathtt{n}_a\},\\
\zeta_b \mapsto \{\mathtt{n}_b\},\\
\zeta_1' \mapsto \{\mathtt{n}_a\},\\
\zeta_1'' \mapsto \{\mathtt{n}_b\},\\
\zeta_1 \mapsto \{\mathtt{n}_a, \mathtt{n}_b\},\\
\zeta_g' \mapsto \{\mathtt{n}_a, \mathtt{n}_b\},\\
\zeta_g \mapsto \{\mathtt{n}_a, \mathtt{n}_b, \mathtt{n}_g\},\\
\zeta_2 \mapsto \{\mathtt{n}_g, \mathtt{n}_a, \mathtt{n}_b\},\\
\zeta_2' \mapsto \{\mathtt{n}_g, \mathtt{n}_a, \mathtt{n}_b\},\\
\zeta_2'' \mapsto \{\mathtt{n}_g, \mathtt{n}_a, \mathtt{n}_b\},\\
\zeta_3 \mapsto \{\mathtt{n}_f, \mathtt{n}_g, \mathtt{n}_a, \mathtt{n}_b\}
\end{array}$$

5.4 Related Work

Subtyping [Cardelli88] adds flexibility to type systems by allowing type coercions to be performed if a type mismatch occurs. It relaxes the form of the type constraints from equations $X = Y$ to inclusions $X \subseteq Y$. The subtyping problem is reduced to the question of whether a system of inclusion constraints has a solution. However there have been no general results on solving inclusion constraints. The algorithms for type inference based on solving inclusion constraints are quite restrictive [Wand87, Stansifer88, Aiken93, Fuh88, Benjamin92]. The subtyping effect system limites the subtype to a subsumption relation on effects. If classical types of expressions are known, the subtyping-based type and effect inference is reduced to solving effect constraints.

Subtyping in effect systems has been previously introduced in explicitly typed languages [Gifford87, Consel93]. There, a subsumption rule similar to the one presented above was used, but since only type checking was performed, its treatment was simpler than ours. The subtyping approach introduced in my thesis shows that type and effect reconstruction may be performed in an implicitly typed language.

Previous effect systems [Talpin92-1, Dornic91, Tang92] have used *subeffecting* that forces a function to have an identical type in different call contexts. My thesis introduces subtyping that allows the same function to have different types for different function calls, as long as they obey certain subtype relation. This *subtyping effect system* avoids effect information to be merged together when forcing two types to be identical, thus improving the accuracy of effect systems based on subeffecting.

5.5 Conclusion

We introduced subtyping in effect systems that allows a function to have different types in different call contexts, as long as they obey certain subtype relation. This *subtyping effect system* avoids effect information to be merged together when forcing two types to be identical, thus collecting more precise effect information than the previous effect system based on subeffecting. We defined the subtype relation, presented the subtyping rule in the static semantics, discussed the reconstruction algorithms $\mathcal{S}$ and proved it sound and complete w.r.t. the static semantics.

Chapter 6

Separate Abstract Interpretation

Type and effect information can be used to extend abstract interpretation in the context of separate compilation.

6.1 Introduction

Abstract interpretation is another basic method for performing static analysis of programs. It is based on denotational semantics by approximating the fixpoint nature of the language semantics. If the abstract interpretation approach performs more precise static analysis due to its more operational nature, effect systems support separate compilation more naturally via module signatures. We introduce the new notion of *separate abstract interpretation* that combines two approaches in a single framework. It extends abstract interpretation in the context of separate compilation based on the type and effect information of module signatures. Types, enriched with effect information, are used to conservatively approximate abstract values of the free variables of programs, thus enabling abstract interpretation to be performed on non-closed expressions. We use control-flow analysis as a motivating example of this new idea.

Shivers' thesis [Shivers91] presents a control-flow analysis based on abstract interpretation. Since a CPS form is used as intermediate representation of programs, control transfers are uniformly represented as tail-recursive function calls. Since this abstract interpretation approach is built on the fixpoint approximation technique, it can distinguish between call environments, thus allowing precise control-flow analysis. Nevertheless fixpoint approximation requires the syntax of function bodies to be known and thus fails to support separate compilation.

Separate abstract interpretation use type and control-flow information to approximate abstract values of free variables in an expression. The basis of the separate abstract interpretation approach is that the abstract semantics of control-flow analysis defined in Shivers' thesis is consistent with the effect semantics used in effect systems. Types describe the structure of values. In particular, from the latent definition d of

a function type $t' \xrightarrow{d} t$, one can determine the set of functions this type may correspond to, together with their control-flow behavior. From such types, one can define conservative approximations of abstract values which are used to pursue the abstract interpretation. We have proved that this new static system conservatively extends the abstract interpretation system and retains all its properties. The control-flow analysis expressed by separate abstract interpretation is as precise as the abstract interpretation approach on closed expressions, but is also able to tackle expressions with free variables by using type and control-flow information to approximate their abstract values.

In the remainder of the chapter, we give the syntax and semantics of CPS programs (Section 6.2), describe an abstract interpretation for control-flow analysis (Section 6.3), adapt the type and effect system of Chapter 4 to CPS programs (Section 6.4), show how these two techniques can be merged together (Section 6.5) to perform separate abstract interpretation (Section 6.6), discuss optimizations for increasing its flexibility and accuracy (Section 6.7), discuss the related work (Section 6.8), before concluding (Section 6.9). All proofs are presented in Appendix 4.

6.2 Dynamic Semantics

6.2.1 CPS Syntax

Since Shivers's abstract interpretation approach uses CPS-transformed programs, we need to define an extended syntax for CPS programs. The main difference with the language defined in Chapter 1 is the introduction of binary functions (to deal with continuation arguments) and the restriction of arguments to self-evaluating expressions.

$$
\begin{array}{lll}
\mathtt{a} ::= & \mathtt{x} & \textit{value identifier} \\
& (\lambda_{\mathtt{n}}\ (\mathtt{x}\ \mathtt{k})\ \mathtt{e}) & \textit{user-defined function} \\
& (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) & \textit{continuation function} \\
 & & \\
\mathtt{e} ::= & (\mathtt{a}\ \mathtt{a}'\ \mathtt{a}'')_{\mathtt{l}} & \textit{function application} \\
& (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} & \textit{continuation application}
\end{array}
$$

User-defined functions are always binary, while continuation functions are unary. In the sequel, without loss of generality, we only specify the semantics of unary functions and calls. By convention, we use `k` as identifiers of continuation functions.

6.2.2 Definition

The dynamic semantics not only defines the values of expressions, but also keeps track of control-flow information during evaluation. We restrict the presentation of the dynamic semantics to CPS expressions.

Following [Shivers91], we use the notion of contours to keep track of scoping information. A *contour* b is a list of labels of function calls describing the current call path. A *contour environment* (also called a *call environment*) β maps any variable to the call path that precedes its actual value binding. A *value* v is either an integer or a closure. A *closure* cl is composed of a lambda expression (including the function label, argument name and function body) and contour environment. A *binding environment*

E is a finite map from pairs of identifiers and contours to values, recording the bindings of identifiers in a given contour.

b	$\in$ Contour	$=$ LCall*	*contour*
β	$\in$ ContourEnv	$=$ Id $\mapsto$ Contour	*contour environment*
v	$\in$ Value	$=$ Int $+$ Closure	*value*
cl	$\in$ Closure	$=$ Fun $*$ ContourEnv	*closure*
E	$\in$ Binding	$=$ Id $*$ Contour $\mapsto$ Value	*binding environment*

The *control-flow information* records the set of functions called at a given call environment. It is defined as a set of tuples $\{(\mathtt{l}, \beta, s)\}$ where the functions in s are called at call site $\mathtt{l}$ and the call environment β; we write such tuples $\{(\mathtt{l}, \beta) \rightsquigarrow s\}$. In the dynamic semantics, this set is always a singleton. We use sets to be compatible with the subsequent non-standard semantics, where they usually have more than one element. The emptyset $\emptyset$ indicates the absence of control-flow information.

$$c \in \text{Control} = \mathcal{P}(\text{LCall} * \text{ContourEnv} * \mathcal{P}(\text{LFun})) \quad \textit{control-flow information}$$

The dynamic semantics is specified by a set of inference rules [Plotkin81]. The usual value environment is split in Shivers' approach in two components: a contour environment β and a binding environment E. The purpose of this uncoupling is to separate the syntactic component of closures from their semantic aspect. This is of outmost importance when performing abstract interpretation where this syntactic component is furthermore restricted to finite expressions.

The inference rule $\beta, E \vdash \mathtt{a} \rightarrow v$ associates the argument $\mathtt{a}$ in the contour environment β and global binding environment E with the value v it evaluates to. In the (var) rule, the value of $\mathtt{x}$ is retrieved from the binding environment E according to the contour where it was bound (recorded by the current contour environment β). In the (abs) rule, the closure is built with its lambda expression and current contour environment. Note that since abstraction expressions are uniquely identified by their function label, we use the function label $\mathtt{n}$ instead of the lambda expression.

$$(var) : \beta, E \vdash \mathtt{x} \rightarrow E(\mathtt{x}, \beta(\mathtt{x}))$$

$$(abs) : \beta, E \vdash \mathtt{n} \rightarrow (\mathtt{n}, \beta)$$

The inference rule $b, \beta, E \vdash \mathtt{e} \rightarrow v, c$ associates the function application $\mathtt{e}$ in the current contour b, contour environment β and global binding environment E with (1) the value v it evaluates to and (2) the control-flow information c recording the control-flow traces during its evaluation. In the (app) rule, control reaches the function call site $\mathtt{l}$ in the contour environment β, binding environment E and contour b, where the function $\mathtt{n}$ is called. Then control enters the function body $\mathtt{e}$, whose control-flow information is c. Note that the binding environment E is global to the whole expression evaluation.

$$(app) : \frac{\begin{array}{l}\beta, E \vdash \mathtt{a} \rightarrow (\lambda_{\mathtt{n}}(\mathtt{x})\ \mathtt{e}, \beta') \\ \beta, E \vdash \mathtt{a}' \rightarrow v' \\ b' = b.\mathtt{l} \\ b', \beta'[\mathtt{x} \mapsto b'], E \vdash \mathtt{e} \rightarrow v, c \\ E(\mathtt{x}, b') = v'\end{array}}{b, \beta, E \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \rightarrow v, c \cup \{(\mathtt{l}, \beta) \rightsquigarrow \{\mathtt{n}\}\}}$$

6.3 Abstract Interpretation Semantics

In Shivers' thesis, first-order control-flow analysis (1CFA) is performed with an abstract interpretation. The contour of the dynamic semantic, which is a call path, is abstracted to a single call site, which is the last element of the call path. Shivers uses a denotational approach for specifying his analysis; we give here a new presentation of this technique using an operational framework which allows us to merge it nicely with the type and effect approach (see Section 6.5).

6.3.1 Definition

The abstract domains correspond to those in the dynamic semantics, except that, since control-flow analysis only deals with functions and ignores integers, values are abstracted to sets of abstract closures. The empty set $\emptyset$ represents any integer.

$\hat{b}$	$\in \widehat{\text{Contour}}$	$= \text{LCall}$	*contour*
$\hat{\beta}$	$\in \widehat{\text{ContourEnv}}$	$= \text{Id} \mapsto \widehat{\text{Contour}}$	*contour environment*
$\hat{v}$	$\in \widehat{\text{Value}}$	$= \mathcal{P}(\widehat{\text{Closure}})$	*value*
$\widehat{cl}$	$\in \widehat{\text{Closure}}$	$= \text{Fun} * \widehat{\text{ContourEnv}}$	*closure*
$\hat{E}$	$\in \widehat{\text{Binding}}$	$= \text{Id} * \widehat{\text{Contour}} \mapsto \widehat{\text{Value}}$	*binding environment*
$\hat{c}$	$\in \widehat{\text{Control}}$	$= \mathcal{P}(\text{LCall} * \widehat{\text{ContourEnv}} * \mathcal{P}(\text{LFun}))$	*control-flow information*

The inference rule $\hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \hat{v}$ associates the argument $\mathtt{a}$ in the contour environment $\hat{\beta}$ and global binding environment $\hat{E}$ with the value $\hat{v}$ it evaluates to.

$$(var) : \hat{\beta}, \hat{E} \vdash \mathtt{x} \rightarrow \hat{E}(\mathtt{x}, \hat{\beta}(\mathtt{x}))$$

$$(abs) : \hat{\beta}, \hat{E} \vdash \mathtt{n} \rightarrow \{(\mathtt{n}, \hat{\beta})\}$$

The inference rule $\hat{\beta}, \hat{E} \vdash \mathtt{e} \rightarrow \hat{v}, \hat{c}$ associates the function application $\mathtt{e}$ in the contour environment $\hat{\beta}$ and global binding environment $\hat{E}$ with (1) the value $\hat{v}$ it evaluates to and (2) the control-flow information $\hat{c}$. In the (app) rule, when control reaches the function call site $\mathtt{l}$ in the contour environment $\hat{\beta}$ and binding environment $\hat{E}$, the function $\mathtt{a}$ is evaluated to a set of closures, while the actual argument $\mathtt{a}'$ is evaluated to its value $\hat{v}'$. Each function $\mathtt{n}_i$ is possibly called at $\mathtt{l}$ at the call environment $\hat{\beta}$ from which control transfers to its function body $\mathtt{e}_i$. Note that, compared to the dynamic semantics, the call path $\hat{b}'$ is limited to a single call site; so, calls to the same function but in different environments in the dynamic semantics may get merged together.

$$(app) : \frac{\left.\begin{array}{l} \hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\} \\ \hat{\beta}, \hat{E} \vdash \mathtt{a}' \rightarrow \hat{v}' \\ \hat{b}' = \mathtt{l} \\ \hat{\beta}_i'[\mathtt{x}_i \mapsto \hat{b}'], \hat{E} \vdash \mathtt{e}_i \rightarrow \hat{v}_i, \hat{c}_i \\ \hat{E}(\mathtt{x}_i, \hat{b}') = \hat{v}' \end{array}\right\} i = 1 \ldots r}{\hat{\beta}, \hat{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \rightarrow \cup_{i=1}^{r} \hat{v}_i, \cup_{i=1}^{r} (\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})}$$

where $[\mathtt{x}_i \mapsto v_i \mid i = 1..n]$ is shorthand for $[][\mathtt{x}_1 \mapsto v_1]...[\mathtt{x}_n \mapsto v_n]$ and $[]$ is the empty constant function.

6.3.2 Correctness

Since the abstract interpretation semantics is defined on finite domains, it terminates. We prove it is well-formed and consistent w.r.t. the dynamic semantics.

Contour environments $\hat{\beta}$, global binding environments $\hat{E}$ and abstract values $\hat{v}$ are related. We define a well-formedness relation $\mathcal{WF}$ between them that ensures that free variables of abstract closures are appropriately bound:

Definition 6.1 (Well-Formedness)

$$\begin{array}{lcl} \mathcal{WF}(\hat{v},\hat{E}) & \Leftrightarrow & \forall(\mathtt{n},\hat{\beta})\in\hat{v},\ \ \mathcal{WF}(\hat{\beta},\hat{E}) \\ \mathcal{WF}(\hat{\beta},\hat{E}) & \Leftrightarrow & \forall \mathtt{x}\in \mathit{Dom}(\hat{\beta}),\ \ (\mathtt{x},\hat{\beta}(\mathtt{x}))\in \mathit{Dom}(\hat{E})\ \wedge\ \mathcal{WF}(\hat{E}(\mathtt{x},\hat{\beta}(\mathtt{x})),\hat{E}) \end{array}$$

Using this definition, we prove that the abstract interpretation semantics is well-formed.

Lemma 6.1 *If $\hat{\beta},\hat{E}\vdash \mathtt{a}\to\hat{v}$ and $\mathcal{WF}(\hat{\beta},\hat{E})$, then $\mathcal{WF}(\hat{v},\hat{E})$.*

Proof *By direct application of Definition 6.1.*

Theorem 6.1 (Well-Formedness of Abstract Semantics) *If $\hat{\beta},\hat{E}\vdash \mathtt{e}\to\hat{v},\hat{c}$ and $\mathcal{WF}(\hat{\beta},\hat{E})$, then*

- *$(\hat{v},\hat{E})$ is well-formed.*
- *All $(\hat{\beta}',\hat{E}')$ used in the $\to$ derivation tree of* e *are well-formed.*

We define the $\leq$ relation as an approximation relation between abstract values: $(\hat{v},\hat{E})\leq(\hat{v}',\hat{E}')$ if $(\hat{v}',\hat{E}')$ is a conservative approximation of $(\hat{v},\hat{E})$. This relation can be straightforwardly extended to compare exact and abstract values.

Definition 6.2 (Consistency of Abstract Values) *For the well-formed $(\hat{v},\hat{E})$ and $(\hat{v}',\hat{E}')$,*

$$\begin{array}{lcl} (\hat{v},\hat{E})\leq(\hat{v}',\hat{E}') & \Leftrightarrow & \forall(\mathtt{n},\hat{\beta})\in\hat{v},\ \exists\hat{\beta}',\ \text{s.t. } (\mathtt{n},\hat{\beta}')\in\hat{v}'\ \wedge\ (\hat{\beta},\hat{E})\leq(\hat{\beta}',\hat{E}') \\ (\hat{\beta},\hat{E})\leq(\hat{\beta}',\hat{E}') & \Leftrightarrow & \forall \mathtt{x}\in\mathit{Dom}(\hat{\beta}),\ \mathtt{x}\in\mathit{Dom}(\hat{\beta}')\ \wedge\ (\hat{E}(\mathtt{x},\hat{\beta}(\mathtt{x})),\hat{E})\leq(\hat{E}'(\mathtt{x},\hat{\beta}'(\mathtt{x})),\hat{E}') \end{array}$$

We next define the $\sqsubseteq$ relation as an approximation relation between abstract control-flow effects: $c\sqsubseteq c'$ if c' is a conservative approximation of c. In other words, c is a more precise control-flow information than c'.

Definition 6.3 (Accuracy of Abstract Effects)

$$c\sqsubseteq c' \ \Leftrightarrow\ \forall(\mathtt{l},\hat{\beta})\rightsquigarrow s\in c,\ \exists\hat{\beta}',s'\ \text{ s.t. } (\mathtt{l},\hat{\beta}')\rightsquigarrow s'\in c'\ \wedge\ s\subseteq s'$$

Using the previous definitions, we can express that the abstract semantics safely approximates the dynamic one for both arguments and expressions:

Lemma 6.2

$$\left.\begin{array}{l}\beta,E\vdash \mathtt{a}\to v\\ \hat{\beta},\hat{E}\vdash \mathtt{a}\to\hat{v}\\ (\beta,E)\leq(\hat{\beta},\hat{E})\end{array}\right\}\ \Rightarrow\ \ (v,E)\leq(\hat{v},\hat{E})$$

Proof *By direct application of Definition 6.2.*

Theorem 6.2 (Consistency of Abstract Semantics)

$$\left.\begin{array}{l}b,\beta,E\vdash \mathtt{e}\to v,c\\ \hat{\beta},\hat{E}\vdash \mathtt{e}\to\hat{v},\hat{c}\\ (\beta,E)\leq(\hat{\beta},\hat{E})\end{array}\right\}\ \Rightarrow\ \left\{\begin{array}{l}(v,E)\leq(\hat{v},\hat{E})\\ c\sqsubseteq\hat{c}\end{array}\right.$$

6.4 Effect System Semantics

We designed in Chapter 4 an effect system to perform 0th-order control-flow analysis in which all call environments are collapsed together. We adapt below this system to CPS expressions.

6.4.1 Definition

A type t can either be the basic type int, a user-defined function type $(t' * t'') \xrightarrow{d} t$ or continuation function type $t' \xrightarrow{d} t$. The *latent definition* d is a set of possibly aliased functions $\mathtt{n}_i$ of the same data type, together with their control-flow effect $\bar{c}_i$. A type environment $\mathcal{E}$ is a finite map from identifiers to types.

$$\begin{array}{llll} d & \in \text{Def} & = \{(\mathtt{n}, \bar{c})\} \mid d' \cup d & \textit{function definition} \\ t & \in \text{Type} & = int \mid (t' * t'') \xrightarrow{d} t \mid t' \xrightarrow{d} t & \textit{type} \\ \mathcal{E} & \in \text{TEnv} & = \text{Id} \mapsto \text{Type} & \textit{type environment} \\ \bar{c} & \in \widehat{\text{Control}} & & \end{array}$$

The control-flow effect $\bar{c}$ of an expression records all the function calls that possibly occur during its evaluation. Since this type and effect semantics does not keep track of call environments, all contour environments that appear in the domain of control-flow effects are unknown, and thus denoted by the empty constant function [].

The inference rule $\mathcal{E} \vdash \mathtt{a} : t$ associates the argument $\mathtt{a}$ in the type environment $\mathcal{E}$ with its type t. In the (abs) rule, the function label $\mathtt{n}$ paired with its control-flow effect $\bar{c}$ is added to the latent definition d of the function type. These rules use implicit subeffecting by adding more functions to d, thus allowing functions of the same data type to be gathered together. This can be used whenever a type mismatch occurs in an application.

$$(var) : \mathcal{E} \vdash \mathtt{x} : \mathcal{E}(\mathtt{x})$$

$$(abs) : \frac{\mathcal{E}_{\mathtt{x}}[\mathtt{x} \mapsto t'] \vdash \mathtt{e} : t, \bar{c} \quad (\mathtt{n}, \bar{c}) \in d}{\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t' \xrightarrow{d} t}$$

The inference rule $\mathcal{E} \vdash \mathtt{e} : t, \bar{c}$ associates the function application $\mathtt{e}$ with its type t and control-flow effect $\bar{c}$. In the (app) rule, the latent definition of the function type is used to determine all the functions $\mathtt{n}$ possibly called at the call site $\mathtt{l}$ and their possible control-flow effect $\bar{c}$.

$$(app) : \frac{\mathcal{E} \vdash \mathtt{a} : t' \xrightarrow{d} t \quad \mathcal{E} \vdash \mathtt{a}' : t'}{\mathcal{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} : t, \cup_{(\mathtt{n}, \bar{c}) \in d} (\bar{c} \cup \{(\mathtt{l}, []) \rightsquigarrow \{\mathtt{n}\}\})}$$

6.4.2 Correctness

We prove that the type and effect semantics is a conservative approximation of the abstract semantics, which means that the abstract interpretation performs more precise control-flow analysis than the effect system.

To define the consistency between the abstract interpretation and the effect system, we introduce the ":" relation between abstract values, abstract environments and types, noted as $(\hat{v}, \hat{E}) : t$. This can be easily extended to environments.

Definition 6.4 (Types of Abstract Values) *For the well-formed* $(\hat{v}, \hat{E})$,

$$\begin{array}{lcl} (\emptyset, \hat{E}) : int & & \\ (\hat{v}, \hat{E}) : t & \Leftrightarrow & \forall(\mathtt{n}, \hat{\beta}) \in \hat{v},\ \exists \mathcal{E},\ \text{s.t.}\ (\hat{\beta}, \hat{E}) : \mathcal{E}\ \wedge\ \mathcal{E} \vdash \mathtt{n} : t \\ (\hat{\beta}, \hat{E}) : \mathcal{E} & \Leftrightarrow & \forall \mathtt{x} \in Dom(\hat{\beta}),\ \mathtt{x} \in Dom(\mathcal{E})\ \wedge\ (\hat{E}(\mathtt{x}, \hat{\beta}(\mathtt{x})), \hat{E}) : \mathcal{E}(\mathtt{x}) \end{array}$$

Using these definitions, we can express that the type semantics conservatively approximates the abstract semantics for both arguments and expressions.

Lemma 6.3

$$\left.\begin{array}{l} \mathcal{E} \vdash \mathtt{a} : t \\ \hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \hat{v} \\ (\hat{\beta}, \hat{E}) : \mathcal{E} \end{array}\right\} \ \Rightarrow\ \ (\hat{v}, \hat{E}) : t$$

Proof *By direct application of Definition 6.4.*

Theorem 6.3 (Types of Abstract Semantics)

$$\left.\begin{array}{l} \mathcal{E} \vdash \mathtt{e} : t, \bar{c} \\ \hat{\beta}, \hat{E} \vdash \mathtt{e} \rightarrow \hat{v}, \hat{c} \\ (\hat{\beta}, \hat{E}) : \mathcal{E} \end{array}\right\} \ \Rightarrow\ \left\{\begin{array}{l} (\hat{v}, \hat{E}) : t \\ \hat{c} \sqsubseteq \bar{c} \end{array}\right.$$

6.5 Approximating Abstract Values

As stated before, control-flow analysis by abstract interpretation is more precise than the one based on the type and effect inference system since it distinguishes between call environments. It however fails to support separate compilation because the value environments $\hat{\beta}$ and $\hat{E}$ are unknown for separately compiled expressions. Note that the type environment $\mathcal{E}$ would be available in this setting.

6.5.1 Approximation Function $\mathcal{A}$

The key idea is to determine a priori the unknown abstract value environment from the type environment, therefore extending the abstract interpretation technique to support separate compilation. The approximation function $\mathcal{A}$ takes a type t and returns its abstract value $\hat{v}$, along with a binding environment $\hat{E}$ that binds the free variables of $\hat{v}$. Abstract closures are thus either built from actual function definitions or approximated from function types.

The type int denotes integers; its abstract value is thus $\emptyset$ and its binding environment $[]$.

$$\mathcal{A}(int) = \quad (\emptyset, [])$$

The function type $(t' * t_0) \xrightarrow{d} t_1$, where d is $\{(\mathtt{n}_i, \bar{c}_i) \mid i = 1 \ldots q\}$, describes a set of user-defined functions $\mathtt{n}_i$ with their control-flow effect $\bar{c}_i$ possibly occurring when

calling $\mathtt{n}_i$. Since the program is in CPS form, t_0 is a continuation type $t \xrightarrow{d'} t_1$ where t is the type of the result value passed to the final continuation. Thus the abstract value $\hat{v}'$ corresponding to the function type is a set of closures $\{(\lambda_{\mathtt{n}_i}(\mathtt{x}\ \mathtt{k}_i)\ \mathtt{e}_i), \hat{\beta}_i \mid i = 1 \ldots q\}$ in which the body $\mathtt{e}_i$ simulates the control-flow effect $\bar{c}_i$ and the contour environment $\hat{\beta}_i$ binds a fresh variable $\mathtt{x}_i$ to a fresh coutour $\mathtt{l}_i$. The binding environment $\hat{E}'$ corresponding to the function type maps the pair $\mathtt{x}_i$ and $\mathtt{l}_i$ to the abstract value $\hat{v}$ corresponding to the return type t. By binding $\mathtt{x}_i$ to $\hat{v}$ in $\hat{E}'$ and applying, in $\mathtt{e}_i$, the final continuation $\mathtt{k}_i$ to $\mathtt{x}_i$ (see below), the abstract value $\hat{v}$ of the result type is passed to its final continuation $\mathtt{k}_i$.

$$
\begin{array}{lll}
\mathcal{A}((t' * t_0) \xrightarrow{d} t_1) = & \textit{let} & \{\mathtt{x}_i\},\ \{\mathtt{l}_i\}\ \textit{and}\ \mathtt{l}\ \textit{fresh} \\
 & & \quad t_0 = t \xrightarrow{d'} t_1 \\
 & & \quad (\hat{v}, \hat{E}) = \mathcal{A}(t) \\
 & & \quad \{\mathtt{e}_i\} = \{\mathcal{S}(\bar{c}_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l})\} \\
 & & \quad \hat{v}' = \{(\lambda_{\mathtt{n}_i}(\mathtt{x}\ \mathtt{k}_i)\ \mathtt{e}_i, [\mathtt{x}_i \mapsto \mathtt{l}_i]) \mid i = 1 \ldots q\} \\
 & & \quad \hat{E}' = \hat{E}[(\mathtt{x}_i, \mathtt{l}_i) \mapsto \hat{v} \mid i = 1 \ldots q] \\
 & \textit{in} & (\hat{v}', \hat{E}') \\
 & \textit{where} & \\
 & d = & \{(\mathtt{n}_i, \bar{c}_i) \mid i = 1 \ldots q\} \\
 & \bar{c}_i = & \{(\mathtt{l}_j, []) \rightsquigarrow \{\mathtt{n}_{j1} \ldots \mathtt{n}_{jr}\} \mid j = 1 \ldots s\}
\end{array}
$$

Each closure body $\mathtt{e}_i$ simulates the control-flow effect $\bar{c}_i$ where, for each call site $\mathtt{l}_j$, all $\mathtt{n}_{jk}$ functions may be called. The expression $\mathcal{S}(\bar{c}, \mathtt{k}, \mathtt{x}, \mathtt{l})$ simulates the control-flow effect $\bar{c}$ and, eventually, applies the continuation $\mathtt{k}$ to the result $\mathtt{x}$ at call site $\mathtt{l}$. It is defined by induction on control-flow effects as below:

$$
\begin{array}{ll}
\mathcal{S}([], \mathtt{k}, \mathtt{x}, \mathtt{l}) & = (\mathtt{k}\ \mathtt{x})_{\mathtt{l}} \\
\mathcal{S}(\bar{c}' \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_1 \ldots \mathtt{n}_r\}\}, \mathtt{k}, \mathtt{x}, \mathtt{l}) & = \mathcal{S}'(\{\mathtt{n}_1 \ldots \mathtt{n}_r\}, \bar{c}', \mathtt{l}', \mathtt{k}, \mathtt{x}, \mathtt{l}) \\
 & \\
\mathcal{S}'(\emptyset, \bar{c}', \mathtt{l}', \mathtt{k}, \mathtt{x}, \mathtt{l}) & = \mathcal{S}(\bar{c}', \mathtt{k}, \mathtt{x}, \mathtt{l}) \\
\mathcal{S}'(s' \cup \{\mathtt{n}'\}, \bar{c}', \mathtt{l}', \mathtt{k}, \mathtt{x}, \mathtt{l}) & = (\ (\lambda_{\mathtt{n}'}(\mathtt{k}')\ \mathcal{S}'(s', \bar{c}', \mathtt{l}', \mathtt{k}, \mathtt{x}, \mathtt{l}))\ \mathtt{k}\ )_{\mathtt{l}'} \\
 & \quad \textit{where}\ \mathtt{k}'\ \textit{is fresh}
\end{array}
$$

At each call site $\mathtt{l}'$ in $\bar{c}$, the function $\mathcal{S}$ calls $\mathcal{S}'$ which is recursively defined on the set of functions $\{\mathtt{n}_1 \ldots \mathtt{n}_r\}$ possibly called at $\mathtt{l}'$. Simulating the behavior of $\bar{c}$ may require replicating call site labels; this is nonetheless acceptable here since this abstract value is automatically generated.

This general definition of $\mathcal{S}$ being somewhat notationally confusing, we give below an example of a closure body for the simple control-flow effect $\bar{c}$:

$$\bar{c} = \{(\mathtt{l}_1, []) \rightsquigarrow \{\mathtt{n}_1\}, (\mathtt{l}_2, []) \rightsquigarrow \{\mathtt{n}_2, \mathtt{n}_3\}\}$$

where the number of call sites is limited to two, and each call site can only call one or two functions. The corresponding closure body $\mathcal{S}(\bar{c}, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l})$ is then:

$$
\begin{array}{l}
(\ (\lambda_{\mathtt{n}_1}(\mathtt{k}_1) \\
\quad (\ (\lambda_{\mathtt{n}_2}\ (\mathtt{k}_2) \\
\quad\quad (\ (\lambda_{\mathtt{n}_3}\ (\mathtt{k}_3) \\
\quad\quad\quad\quad (\mathtt{k}_i\ \mathtt{x}_i)_{\mathtt{l}}) \\
\quad\quad\ \mathtt{k}_i\ )_{\mathtt{l}_2}) \\
\quad\ \mathtt{k}_i\ )_{\mathtt{l}_2}) \\
\mathtt{k}_i\ )_{\mathtt{l}_1}
\end{array}
$$

6.5.2 Correctness of $\mathcal{A}$

The approximation function $\mathcal{A}$ has the following properties :

Lemma 6.4 (Well-Formedness of $\mathcal{A}(t)$) *$\mathcal{A}(t)$ is well-formed.*

Note that the abstract values $\hat{v}'$ defined by $\mathcal{A}$ include simulated call environments whose domains contain only fresh variables. We thus extend the approximation relation $\leq$ to compare the abstract values and the approximated ones in the following way:

Definition 6.5 (Consistent Abstract Values) *For the well-formed $(\hat{v}, \hat{E})$ and $(\hat{v}', \hat{E}')$, if $(\hat{v}', \hat{E}')$ is defined via $\mathcal{A}$, then*

$$(\hat{v}, \hat{E}) \leq (\hat{v}', \hat{E}') \quad \Leftrightarrow \quad \forall (\mathtt{n}, \hat{\beta}) \in \hat{v},\ \exists \hat{\beta}',\ \text{s.t. } (\mathtt{n}, \hat{\beta}') \in \hat{v}'$$

Using this extended definition, we get:

Lemma 6.5 (Consistency of $\mathcal{A}(t)$) *If $(\hat{v}, \hat{E}) : t$, then $(\hat{v}, \hat{E}) \leq \mathcal{A}(t)$*

Since simulated call environments do not correspond to actual call environments, we define, for the purpose of comparing them, a function $\mathcal{D}$ that deletes these simulated environments in the control-flow effects obtained by abstract interpretation.

$$
\begin{array}{ll}
\mathcal{D}([]) & = [] \\
\mathcal{D}(\hat{c} \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow s\}) & = \mathcal{D}(\hat{c}) \cup \{(\mathtt{l}, []) \rightsquigarrow s\}
\end{array}
$$

Using the initial identity continuation Id at a given call site $\mathtt{l}_k$, the abstract interpretation of any of the q expressions $\mathtt{e}_i$, built by the function $\mathcal{S}$ from the control-flow effect $\bar{c}_i$ given by the type semantics, yields a control-flow effect $\hat{c}_i$ which, modulo $\mathcal{D}$, is the *same* as $\bar{c}_i$.

Lemma 6.6 (Simulation) *For any $\hat{\beta}_1$ and $\hat{E}_1$, if*

$$\hat{\beta}_1[\mathtt{x}_i \mapsto \mathtt{l}_i][\mathtt{k}_i \mapsto \mathtt{l}_k], \hat{E}_1[(\mathtt{x}_i, \mathtt{l}_i) \mapsto \hat{v}][(\mathtt{k}_i, \mathtt{l}_k) \mapsto \{Id\}] \vdash \mathcal{S}(\bar{c}_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \rightarrow \hat{v}, \hat{c}_i$$

then $\mathcal{D}(\hat{c}_i) = \bar{c}_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\}$.

6.6 Separate Abstract Interpretation

Separate abstract interpretation uses types and effects to compute conservative approximations of abstract values of the free variables occurring in a separately compiled CPS expression $\mathtt{e}$. These values are used to create initial environments in which the classical abstract interpretation is performed. These initial abstract value environments $\hat{\beta}_0$ and $\hat{E}_0$ are defined via the function $\mathcal{A}$, based on the type environment $\mathcal{E}$ of $\mathtt{e}$.

Given a CPS expression $\mathtt{e}$, its initial contour environment $\hat{\beta}_0$ maps free variables to the fresh call site labels, since their actual binding call sites are unknown. Its initial binding environment $\hat{E}_0$ is defined not only on the free variables of $\mathtt{e}$, but also on those introduced by $\mathcal{A}$; these additional identifiers are bound in the additional binding environments $\hat{E}$ given by $\mathcal{A}$.

$$\hat{\beta}_0 = \quad [\mathtt{x} \mapsto \mathtt{l} \mid \mathtt{x} \in \mathit{Dom}(\mathcal{E}) \ \wedge \ \mathit{fresh}\ \mathtt{l}]$$

$$\hat{E}_0 = \quad \cup_{\mathtt{x} \in \mathit{Dom}(\mathcal{E}) \ \wedge \ (\hat{v},\hat{E}) = \mathcal{A}(\mathcal{E}(\mathtt{x}))} \ \hat{E}[(\mathtt{x}, \hat{\beta}_0(\mathtt{x})) \mapsto \hat{v}]$$

where $\cup$ is the function union with the property that $(f \cup g)(\mathtt{x}) = f(\mathtt{x}) \cup g(\mathtt{x})$.

The approximated initial environments have the following properties, corresponding to those of the approximation function $\mathcal{A}$.

Lemma 6.7 (Well-Formedness of $(\hat{\beta}_0, \hat{E}_0)$) *$(\hat{\beta}_0, \hat{E}_0)$ is well-formed.*

Lemma 6.8 (Consistency of $(\hat{\beta}_0, \hat{E}_0)$) *If $(\hat{\beta}_0', \hat{E}_0') : \mathcal{E}$, then $(\hat{\beta}_0', \hat{E}_0') \leq (\hat{\beta}_0, \hat{E}_0)$*

Classical abstract interpretation can then simply be applied on $\mathtt{e}$ with these approximated initial environments:

$$\hat{\beta}_0, \hat{E}_0 \vdash \mathtt{e} \rightarrow \hat{v}, \hat{c}$$

to implement the notion of separate abstract interpretation. Thanks to these approximated environments, we extended the abstract interpretation approach to support separate compilation. This new interpretation enjoys all the properties of the abstract interpretation semantics presented above, i.e. it terminates and is well-formed. It is thus a conservative approximation of abstract interpretation.

Theorem 6.4 (Separate Abstract Interpretation) *Separate abstract interpretation is a conservative extension of abstract interpretation.*

6.7 Optimizations

6.7.1 Subtyping Effect Systems

As stated in Chapter 5, subtyping can be introduced to improve the accuracy of effect systems based on subeffecting. Therefore we can extend abstract interpretation with subtyping effect systems to perform more precise control-flow analysis.

6.7.2 Flexibility of Abstract Semantics

The abstract interpretation semantics defined in Section 6.3 restricts a lambda expression $\mathtt{n}$ in the value environment $\hat{\beta}, \hat{E}$ to a singleton $\{(\mathtt{n}, \hat{\beta})\}$, which limits the number of programs derivable by the abstract semantics. To increase the flexibility of the abstract semantics, we could adjust (abs) rule to (abs'), which allows a lambda expression to admit a larger abstract value as long as its type is persevered.

$$(abs'): \quad \frac{\begin{array}{l}(\hat{\beta}, \hat{E}) : \mathcal{E} \\ \mathcal{E} \vdash \mathtt{n} : t \\ \mathcal{WF}(\hat{v}, \hat{E}) \\ (\hat{v}, \hat{E}) : t\end{array}}{\hat{\beta}, \hat{E} \vdash \mathtt{n} \rightarrow \{(\mathtt{n}, \hat{\beta})\} \cup \hat{v}}$$

By direct application of Definition 6.1 and Definition 6.4, we can see that the (abs') rule preserves the properties of (abs), namely (1) well-formedness, i.e. if $\mathcal{WF}(\hat{\beta}, \hat{E})$, then $\mathcal{WF}(\{(\mathtt{n}, \hat{\beta})\} \cup \hat{v}, \hat{E})$ and (2) typability, i.e. $(\{(\mathtt{n}, \hat{\beta})\} \cup \hat{v}, \hat{E}) : t$. Thus this new abstract semantics enjoys all of the properties (see Section 6.3 and Section 6.4) of the previous abstact semantics, but is more flexible. It terminates, is well-formed, is a conservative approximation of the dynamic semantics, and is more precise than the type semantics.

6.7.3 Local Control-Flow Effects

Even though the previously described approximation function $\mathcal{A}$ enables abstract interpretation to be applied in the presence of separate compilation, it has the major drawback of limiting its accuracy. Indeed, in the function types of CPS expressions, the control-flow effects in their latent definitions d represent not only the local control-flow effects of function bodies but also,via final continuation calls, those of the continuation of the program. Consequently, the accuracy of separate abstract interpretation is only as good as the one of the type and effect analysis.

To improve the analysis requires the use of $\mathcal{A}$ on types restricted to local control-flow effects. This can be achieved by computing the abstract values of the free variables on the basis of their direct, non-CPS type in the following way.

Using the previous notations, the continuation type $t_0 = t \xrightarrow{d'} t_1$, where $d' = \{(\mathtt{n}'_i, \bar{c}'_i) \mid i = 1 \ldots p\}$, describes a set of continuation functions $\mathtt{n}'_i$ and their control-flow effect $\bar{c}'_i$. User-defined functions of type $(t' * t_0) \xrightarrow{d} t_1$, where $d = \{(\mathtt{n}_i, \bar{c}_i) \mid i = 1 \ldots q\}$, accept continuations of type t_0, beside the argument of type t'. The control-flow effects $\bar{c}_i$ of their bodies include the control-flow effects that correspond to applying the final continuation to their result. By subtracting this control-flow effect $\cup_{i=1}^{r} \bar{c}'_i$ from $\bar{c}_i$, the remaining effect only corresponds to the local control-flow effect of the body of the function $\mathtt{n}_i$. This is equivalent to the control-flow effect recorded in the corresponding *direct* function type, if one ignores all continuation calls in CPS types.

To summarize, given a non-closed expression $\mathtt{e}$, control-flow analysis using separate abstract interpretation is performed according to the following steps:

1. Apply type and effect inference to get the type environment $\mathcal{E}$ of e.
2. Use the function $\mathcal{A}$ onto $\mathcal{E}$ to approximate the corresponding initial abstract value environment $(\hat{\beta}_0, \hat{E}_0)$.
3. Transform e to its CPS form e′.
4. Apply the classical abstract interpretation algorithm to e′, based on $(\hat{\beta}_0, \hat{E}_0)$, to get the control-flow information.

6.8 Related Work

In Shivers′s thesis [Shivers91], control-flow analyses of arbitrary order (nCFA, where n is the order) on programs written in continuation-passing style (CPS) [Appel89] are defined and performed by using an abstract interpretation approach. These control-flow analyses are able to distinguish different call environments but fail to support separate compilation, thus limiting their real-world application.

Effect systems extend type systems with effect information. Just as types describe the possible values of expressions, effects describe their possible evaluation behaviors. Our previous papers [Tang92, Tang93] presented a type and control-flow effect system where the inferred control-flow effects of expressions describe all control-flow traces possibly occurring during their evaluation. This analysis supports separate compilation but collapses call environments together, thus is less precise.

Here, we extend the abstract interpretation approach for 1CFA to support modularity, i.e. separate compilation, by approximating unknown value environments of expressions via their type environments. Thus our control-flow analysis performs 1CFA, and possibly nCFA, even in the presence of separate compilation.

6.9 Conclusion

This chapter introduced a new technique to extend abstract interpretation approach in the context of separate compilation based on type and effect information. This separate abstract interpretation makes the control-flow analysis as effective as the abstract interpretation approach on closed expressions, but is also able to tackle expressions with free variables, using their type to approximate their abstract value. We proved that the control-flow information obtained by this new analysis is a conservative approximation of abstract interpretation and is more precise than the type and effect system.

Chapter 7

Higher-Order Escape Analysis

Control-flow analysis is helpful to choose an efficient closure allocation strategy.

7.1 Introduction

Escape analysis helps compilers optimize closure allocation in functional program implementation. It determines, at compile time, the free variables of functions that outlive the environment in which they are defined. Therefore non-escaping variables can be safely allocated in the stack while reserving the heap only for escaping ones. We present a new static escape analysis based on the control-flow effect systems (Chapter 3 4, 5). The escape analysis can be performed in presence of higher-order functions, imperative constructs and separate compilation. We design a stack-based abstract machine where closures are allocated in the stack and whenever functions are called, their escaping free variables are copied from the stack to the heap.

In the sequel, we present a static criteria for identifying escaping functions (Section 7.2), design a stack-based abstract machine to show an optimized allocation strategy (Section 7.3) and discuss related work (Section 7.4) before concluding (Section 7.5).

7.2 Identifying Escaping Variables

7.2.1 Escaping Variables

Function values are represented by compilers as closures. Closures are composed of the function code and the free variables that form its environment. Since functions are first-class values, their free variables may outlive the environment in which they are defined. Here we use integers to specify the lexical level of each expression. The top-level expression has the lexical level 0. The escaping variables do not obey the LIFO stack-allocation strategy used in traditional call mechanism and must be heap-allocated. Heap allocation is more general than stack allocation in the sense that heap allocation can be used for all free variables of functions, while stack allocation is only safe for non-escaping ones. However, stack allocation is cheaper than heap allocation

because useless storage is simply reclaimed by updating a pointer instead of calling the garbage collector. Finding an efficient allocation strategy for closure environments is therefore important for optimizing compilers of functional languages. A good strategy of closure allocation for functional languages is to stack allocate non-escaping variables of functions while reserving the more expensive heap allocation to escaping ones. The key problem is thus to identify escaping variables at compile time safely and as precisely as possible.

7.2.2 From Types to Escaping variables

Escaping variables can be identified based on the types and control-flow effects inferred by the previously defined control-flow effect systems (see Chapter 3,4,5). For any type-checked function call (`e e'`) at the lexical level `l`, `e` must have a function type $t' \xrightarrow{c} t$ where t' is the type of the argument `e'` and t is the result type of this function call. If a function `n` is returned as the result of this function call, it must be recorded by t via its latent control-flow effects; If `e'` is evaluated to a memory location loc, t' must be $ref(t'')$. When a function `n` is allocated in loc during the evaluation of `e`, it must be recorded by t'' via its latent control-flow effects. Note that the function type $t' \xrightarrow{c} t$ includes the information that helps identify the escaping variables of functions. More precisely, for each function, we identify its escape-level and escape-set. The *escape-level* is the smallest lexical level at which the function escapes (or *infinity* if the function does not escape) while its *escape-set* is the set of free variables of the function body that are bound within its escape-level and definition site. The variables recorded by the escape-set are escaping variables.

To compute this information, we use two environments, LE and EE. The *lexical environment* LE maps identifiers to the integer lexical levels at which they are bound. The *escape-environment* EE maps a lexical level `l` defining a lambda expression to the function type $t \xrightarrow{c} t'$; this type records, via the latent control-flow effects possibly present in t or t', the names of all of the functions that may escape at the level `l`. A function `n`, defined at the level `l'`, escapes at the level `l` by being either part of the value returned at the level `l` (its name is free in t') or stored in a location bound at the level `l` (its name is free in t); the function `n` is then said to escape from `l'` to `l`.

7.2.3 Algorithm $\mathcal{I}$

Given a lexical environment LE and an escape-environment EE, the algorithm $\mathcal{I}$ updates, for an expression `e` at the level `l`, the identification function i. This *identification function* maps a function name to its escape-set. A function may escape to multiple lexical levels; conservatively, the escape-level is the minimum of them. The escape set records all of the free variables of a function bound at a lexical level larger than the escape-level. These two escape attributes are conservative approximations of their dynamic counterparts.

We assume in $\mathcal{I}$ that the expression is completely typed, the type and control-flow information having been previously inferred by the effect systems.

$$
\begin{array}{l}
\mathcal{I}(\mathtt{e})\ \mathrm{LE}\ \mathrm{EE}\ \mathtt{l}\ i = \\
\quad \mathit{case}\ \mathtt{e} \\
\quad \mathtt{x} \Rightarrow i \\
\quad (\mathtt{e}\ \mathtt{e}') \Rightarrow \mathcal{I}(\mathtt{e})\ \mathrm{LE}\ \mathrm{EE}\ \mathtt{l}\ \ (\mathcal{I}(\mathtt{e}')\ \mathrm{LE}\ \mathrm{EE}\ \mathtt{l}\ i) \\
\quad (\lambda \mathtt{n}\ (\mathtt{x} : t)\ \mathtt{e}' : t') \Rightarrow \\
\qquad \mathtt{let}\ el = \mathit{Min}\{\mathtt{l} \mid \mathtt{n} \in \mathit{fn}(\mathrm{EE}(\mathtt{l}))\} \\
\qquad\quad es = \{\mathtt{y} \in \mathit{fv}(\mathtt{e}) \mid \mathrm{LE}(\mathtt{y}) \geq el\} \\
\qquad\quad i' = \mathcal{I}(\mathtt{e}')\ (\mathrm{LE}\{\mathtt{x} \mapsto \mathtt{l}\}) \\
\qquad\qquad\qquad\quad (\mathrm{EE}\{\mathtt{l} \mapsto t \xrightarrow{\emptyset} t'\}) \\
\qquad\qquad\qquad\quad (\mathtt{l}+1)\ i \\
\qquad \mathrm{in}\ i'\{\mathtt{n} \mapsto es\}
\end{array}
$$

where fv computes the set of free variables of expressions and environments, while fn restricts this set to function names. By convention, $\mathit{Min}\emptyset$ is defined to be *infinity*. The identification function of a whole program expression `p` is given by calling $\mathcal{I}$ on `p` with the empty environment, lexical level 0 and the identity function.

The previously computed escaping variables can be used to efficiently allocate closure environments. For any function, only its escaping variables need to be allocated in the heap; the others can, as before, be allocated in the stack. We introduce an *abstract machine* to show how to use this optimized closure allocation strategy.

7.3 A Stack-based Abstract Machine

7.3.1 Stack Calling Convention

The abstract machine is built upon the stack calling convention. Following compilers of traditional languages [Aho86], an *activation record* is built for each function call, which records the arguments of the function call, together with other information, such as return point, etc. The activation records are allocated in a control stack, obeying the LIFO strategy. The only difference is that when functions are called, their escaping variables, which do not obey the LIFO stack-allocation strategy, have to be *copied* from the control stack (the activation records), to the heap, while all other non-escaping free variables remain in the control stack. These heap allocated environments are called *escaping environments*. Therefore closures in our compiler include two kinds of environments : non-escaping environments recorded in the activation records in the control stack, and escaping environments in the heap. When compiling non-escaping functions, our compiler is as efficient as compiling traditional languages. When escaping functions exist, it is more efficient than compiling other functional languages such as `SML` or `SCHEME` [Krantz87, Steele78, Appel87] where the static escape information is unknown or less precise. We present the structure of the abstract machine, describe the semantics of its instructions, and give an algorithm $\mathcal{C}$ to transform the programs to a list of instructions.

7.3.2 Structure

The state of the abstract machine is specified by the following five elements :

(*Code*, *Accu*, *TempS*, *EsEnv*, *ContS*)

which respectively are :

- User program (*Code*) : it includes a list of instructions of the abstract machine, which are defined in Section 7.3.3.
- Accumulator (*Accu*) : it stores the immediate result values of the abstract machine. The values can be integers or closures. The closure is composed of the code of the function body, its escaping environment recorded by *EsEnv* and non-escaping environment recorded in the activation records ActR;
- Temporary stack (*TempS*) : it temporarily stores the environments of functions. It is organized as a list of values.
- Escaping environment (*EsEnv*) : it records the escaping free variables of a function, which is represented as a vector. Accessing escaping variables is by their indices in the vector. All escaping environments are allocated in the heap.
- Control stack (*ContS*) : it stores the activation records of function calls linked together by the access link. They obey the LIFO allocated strategy. An activation record is composed of the argument of the function call, the access link pointing to another activation record, the return point where control will return at the end of the function call. The *return point* includes the code and its escaping environment. The environments of functions are recorded in the activation records. Accessing them is by their indices in the linked activation records.

7.3.3 Instructions

The abstract machine has a set of instructions:

- stack(i) : accesses the value of a variable in the linked activation records allocated in the control stack *ContS* via the function $\texttt{access}(s, i)$ where i is its index in the linked activation records. The function $\texttt{access}(s, i)$ accesses the value recorded in the ith activation record starting from s.

$$\begin{array}{ll} \texttt{access}(\emptyset, i) & = \text{fail} \\ \texttt{access}(s'.(v, s, r), 1) & = v \\ \texttt{access}(s'.(v, s, r), i) & = \texttt{access}(s, i-1) \end{array}$$

- heap(i) : accesses the value of an escaping variable in the escaping environment *EsEnv* allocated in the heap, where i is its index of in the vector.
- push: pushes the current value on the temporary stack *TempS*.
- close(m,c) : closes a function with the code c of its body, its escaping environment including m elements (poped from the temporary stack *TempS*) and its current activation record (including non-escaping variables).

- call: updates the control stack *ContS* by building a new activation record for the function call. This activation record is formed by the argument of the called function (poped from the temporary stack *TempS*), the access link (copied from the non-escaping environment in the closure), and the return point (including the rest of code and the current escaping environment). The current escaping environment is updated with the one recorded in the closure and then control is transferred to the code of the function body.

- return: updates the control stack *ContS* by poping the activation record, updates the current escaping environment *EsEnv* with the one recorded by the return point of the poped activation record and transfers control to the code recorded by the return point.

The semantics of the abstract machine is operationally given by the state transition [Plotkin81] of each instruction.

Code	*Accu*	*TempS*	*EsEnv*	*ContS*
stack(i) ; c	v	a	e	s
c	access(s, i)	a	e	s
heap(i) ; c	v	a	$e = v_1..v_i..v_n$	s
c	v_i	a	e	s
push ; c	v	a	e	s
c	v	$a.v$	e	s
close(m, c') ; c	v	$a.v_1 \ldots v_m$	e	s
c	$(c', \{v_1 \ldots v_m\}, s)$	a	e	s
call ; c	$v = (c', e', s')$	$a.v'$	e	s
c'	v	a	e'	$s.(v', s', (c, e))$
return ; c	v	a	e	$s.(v', s', (c', e'))$
c'	v	a	e'	s

7.3.4 Translator $\mathcal{C}$

The translator $\mathcal{C}$ transforms an expression to a list of instructions of the abstract machine based on the result of previous escaping analysis.

A stack environment E_s maps identifiers to their indices in the control stack *ContS*. A heap environment E_h maps escaping variables to their indices in the escaping environment *EsEnv*. The identification function i maps all function names in the expression e to their escaping variables. Given an identification function i, a stack environment E_s and a heap environment E_h, the translator $\mathcal{C}$ transforms the expression e at the lexical level l into a list of instructions of the abstract machine.

For a variable, if it is an escaping variable then its value is accessed by its index in the escaping environment, otherwise it is accessed in the control stack.

$$\begin{array}{l}\mathcal{C}[\![\mathtt{x}]\!]\ i\ E_s\ E_h\ l = \\ \quad \mathit{if} \quad \mathtt{x} \in \mathit{Dom}(E_h) \\ \quad \mathit{then}\ \underline{\mathtt{heap}}(E_h(\mathtt{x})) \\ \quad \mathit{else}\ \ \underline{\mathtt{stack}}(l - E_s(\mathtt{x}))\end{array}$$

For an application, the expression in the argument position is compiled and its result is temporarily pushed onto the temporary stack *TempS*. Then the expression in the function position is compiled, by building its closure. Finally the function is called by constructing its activation record in the control stack *ContS*.

$$\begin{array}{l}\mathcal{C}[\![(\mathtt{e}\ \mathtt{e}')]\!]\ i\ E_s\ E_h\ l = \\ \quad \mathcal{C}[\![\mathtt{e}']\!]\ i\ E_s\ E_h\ l\ ;\ \underline{\mathtt{push}}\ ;\ \mathcal{C}[\![\mathtt{e}]\!]\ i\ E_s\ E_h\ l\ ;\ \underline{\mathtt{call}}\end{array}$$

For an abstraction, all its escaping variables (recorded by the identification function i) should be copied from the control stack and temporarily stored in the temporary stack *TempS*. The function is closed with the code of its body, the escaping variables moved from *TempS* and the current control stack pointer.

$$\begin{array}{l}\mathcal{C}[\![(\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e})]\!]\ i\ E_s\ E_h\ l = \\ \quad \mathit{let}\ \ \{\mathtt{x}_1 ... \mathtt{x}_m\} = i(\mathtt{n}) \\ \qquad \{i_1 \ldots i_m\} = \{E_s(\mathtt{x}_1) ... E_s(\mathtt{x}_m)\} \\ \quad \mathit{in}\ \ \underline{\mathtt{stack}}(l - i_1)\ ;\ \underline{\mathtt{push}}\ ; \\ \qquad \vdots \\ \qquad \underline{\mathtt{stack}}(l - i_m)\ ;\ \underline{\mathtt{push}}\ ; \\ \qquad \underline{\mathtt{close}}(m, \mathcal{C}[\![\mathtt{e}]\!]\ i\ (E_s[\mathtt{x} \mapsto l])\ ([][\mathtt{x}_1 \mapsto 1] ... [\mathtt{x}_m \mapsto m])\ (l+1)\ ;\ \underline{\mathtt{return}})\end{array}$$

7.3.5 Example

We use the example `exam` to show how our abstract machine behaves.

$$\mathtt{exam} = (\lambda_{\mathtt{n}_z}\ (\mathtt{z})\ ((\lambda_{\mathtt{n}_x}\ (\mathtt{x})\ (\lambda_{\mathtt{n}_y}\ (\mathtt{y})\ \mathtt{x}))\ \mathtt{z}))$$

After applying previous escape analysis to the program, we get its identification function i which maps the lambda expressions $\mathtt{n}_z$, $\mathtt{n}_y$ and $\mathtt{n}_x$ to their escaping variables, $\emptyset$, $\emptyset$ and $\{\mathtt{x}\}$ respectively. The translator $\mathcal{C}$ starts from the empty initial environments E_s and E_h and the initial lexical level 0.

Since the lambda expression $\mathtt{n}_z$ has no escaping variables, it is closed with the code of its body and the pointer of the current control stack.

$$\begin{array}{l}\mathcal{C}[\![(\lambda_{\mathtt{n}_z}\ (\mathtt{z})\ ((\lambda_{\mathtt{n}_x}\ (\mathtt{x})\ (\lambda_{\mathtt{n}_y}\ (\mathtt{y})\ \mathtt{x}))\ \mathtt{z}))]\!]\ i\ []\ []\ 0 = \\ \quad \underline{\mathtt{close}}(0,\ \mathcal{C}[\![((\lambda_{\mathtt{n}_x}\ (\mathtt{x})\ (\lambda_{\mathtt{n}_y}\ (\mathtt{y})\ \mathtt{x}))\ \mathtt{z})]\!]\ i\ [\mathtt{z} \mapsto 0]\ []\ 1\ ;\ \underline{\mathtt{return}})\end{array}$$

For the function application $((\lambda_{\mathtt{n}_x}\ (\mathtt{x})\ (\lambda_{\mathtt{n}_y}\ (\mathtt{y})\ \mathtt{x}))\ \mathtt{z})$, the variable `z` has to be accessed in the stack by $\underline{\mathtt{stack}}(1)$; Then the lambda expression $\mathtt{n}_x$ is compiled.

$\mathcal{C}[\![((\lambda_{\mathtt{n}_x}\ (\mathtt{x})\ (\lambda_{\mathtt{n}_y}\ (\mathtt{y})\ \mathtt{x}))\ \mathtt{z})]\!]\ i\ [\mathtt{z} \mapsto 0]\ []\ 1 =$
$\quad \mathcal{C}[\![\mathtt{z}]\!]\ i\ [\mathtt{z} \mapsto 0]\ []\ 1$; $\underline{\mathtt{push}}$;
$\quad \mathcal{C}[\![(\lambda_{\mathtt{n}_x}\ (\mathtt{x})\ (\lambda_{\mathtt{n}_y}\ (\mathtt{y})\ \mathtt{x}))]\!]\ i\ [\mathtt{z} \mapsto 0]\ []\ 1$;
$\quad \underline{\mathtt{call}}$;
where $\mathcal{C}[\![\mathtt{z}]\!]\ i\ [\mathtt{z} \mapsto 0]\ []\ 1 = \underline{\mathtt{stack}}(1)$

The code for $\mathtt{n}_x$ is similar to that of $\mathtt{n}_z$.

$\mathcal{C}[\![(\lambda_{\mathtt{n}_x}\ (\mathtt{x})\ (\lambda_{\mathtt{n}_y}\ (\mathtt{y})\ \mathtt{x}))]\!]\ i\ [\mathtt{z} \mapsto 0]\ []\ 1 =$
$\quad \underline{\mathtt{close}}(0,\ \mathcal{C}[\![(\lambda_{\mathtt{n}_y}\ (\mathtt{y})\ \mathtt{x})]\!]\ i\ [\mathtt{z} \mapsto 0, \mathtt{x} \mapsto 1]\ []\ 2\ ;\ \underline{\mathtt{return}})$

Sicnce the lambda expression $\mathtt{n}_y$ has the escaping variable $\mathtt{x}$, $\mathtt{x}$ has to be copied from the control-flow stack *ContS* to the temporary stack *TempS* by performing $\underline{\mathtt{stack}}(1)$; $\underline{\mathtt{push}}$. Then $\mathtt{n}_y$ is closed by the code of its body, the pointer of the current control stack and its escaping environment (popped from the temporary stack *TempS*).

$\mathcal{C}[\![(\lambda_{\mathtt{n}_y}\ (\mathtt{y})\ \mathtt{x})]\!]\ i\ [\mathtt{z} \mapsto 0, \mathtt{x} \mapsto 1]\ []\ 2 =$
$\quad \underline{\mathtt{stack}}(1)$; $\underline{\mathtt{push}}$;
$\quad \underline{\mathtt{close}}(1, \mathcal{C}[\![\mathtt{x}]\!]\ i\ [\mathtt{z} \mapsto 0, \mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2]\ [\mathtt{x} \mapsto 1]\ 3\ ;\ \underline{\mathtt{return}})$
where $\mathcal{C}[\![\mathtt{x}]\!]\ i\ [\mathtt{z} \mapsto 0, \mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2]\ [\mathtt{x} \mapsto 1]\ 3 = \underline{\mathtt{heap}}(1)$

The abstract machine code of the program `exam` is thus as below :

$\underline{\mathtt{close}}(0,\ \underline{\mathtt{stack}}(1)$; $\underline{\mathtt{push}}$;
$\qquad \underline{\mathtt{close}}(0, \underline{\mathtt{stack}}(1)$; $\underline{\mathtt{push}}$; $\underline{\mathtt{close}}(1, \underline{\mathtt{heap}}(1)$; $\underline{\mathtt{return}})$; $\underline{\mathtt{return}})$;
$\qquad \underline{\mathtt{call}}$;
$\qquad \underline{\mathtt{return}})$

7.4 Related Work

Escape information can be identified either at compile time, based on a static analysis of programs, or at run time, using a run-time checking mechanism [Baker92]. Here we only discuss compile-time approaches.

A simple escape analysis was used in the Scheme [Rees88] compilers Rabbit [Steele78] and ORBIT [Krantz87] to optimize closure allocation. These analyses are syntax-based, i.e., the escaping functions are identified by their syntactical context through a recursive walk of expressions. Our analysis is based on the type and control-flow information of expressions, computed by a type and effect inference system, thus is more precise, in particular when dealing with higher-order functions.

A higher-order escape analysis [Goldberg90] on a typed functional language was performed using abstract interpretation. This analysis computes abstract escape functions of the programs based on fixpoint approximations. Our analysis applies an identifying algorithm to the programs based on their types and control-flow information, which makes our analysis process a lower cost; Another benefit of using effect systems is that it can straightforwardly deal with imperative constructs in functional languages thanks to explicit reference types of locations. Another main difference between these

two analyses is the definition of escape information, which decides the way that they are used for optimizing closure allocation. Goldberg′s analysis identifies, for each function call, which arguments possibly outlive this function call. The escaping arguments are allocated immediately in the heap whenever the function call is performed. Our analysis identifies, for each function, the free variables that possibly outlive the environment where they are defined. The escaping free variables have to be *copied* to the heap whenever these functions are called at run time, which makes our system more expansive at run-time application.

Taking the example used in Goldberg's paper, we can see the differences between these two analyses.

```
let f x y z = x + y + z
    g a b = f b a
in  g 1 2
```

In Goldberg's analysis, abstract escape functions for the function f and g are formed by fixed point iteration. With these abstract escape functions, the system identifies if their arguments escape from the calls to them. Here since 1 and 2 escape from the function call (g 1 2), they have to be allocated in the heap when compiling the function call.

In our analysis, the identifying algorithm identifies that x, y are escaping variables of the function $\lambda_{\mathtt{n}_z}\ (z)\ (x\ +\ y\ +\ z)$ from the result type $int \overset{\{\mathtt{n}_z\}}{\rightarrow} int$ of (g 1 2). Whenever the function $\mathtt{n}_z$ is called at run time, their values have to be copied from the stack to the heap.

Finally we compare the accuracy of these two analysis. Generally, the abstract interpretation approach performs more precise analysis than effect systems due to its more operational nature. However, since our escape information is used together with run-time evaluation, only the escaping variables of functions that are reached at run time need to be allocated in the heap, which makes our analysis more precise in some cases, for example, in conditional expressions.

7.5 Conclusion

We presented a new higher-order escape analysis based on type and effect systems, supporting higher-order functions, imperative constructs and separate compilation. The escape analysis determines, at compile time, the free variables of functions that outlive the environment in which they are defined. Based on these compile-time knowledge of escape information, we designed a stack-based abstract machine, where non-escaping variables are safely allocated in the stack and only escaping variables are allocated in the heap.

Conclusion

My thesis work lies on the border of the theory and the practice.

We extend and combine two static analysis approaches – effect systems and abstract interpretation, and study their application in performing control-flow analysis.

We present new control-flow analysis systems based on effect systems. *Subtyping* is introduced to increase the flexibility of effect systems. The subtype relation is defined by a subsumption relation on effect information. A new type and effect reconstruction algorithm is designed, which for each expression already typed with classical types, reconstructs its type and effect based on subtyping. The subtyping approach allows functions to have different types in different call contexts instead of having a unique type, thus improving the accuracy of effect systems based on subeffecting. The current subtyping effect system is built upon a monotonic type system. However it can be extended to ML polymorphic type systems, which makes subtyping effect systems more powerful. Another open issue is how to use the subtyping effect system in the presence of side-effects.

We introduce the new notion of *separate abstract interpretation* which combines effect systems and abstract interpretation in a single framework. By approximating abstract values of free variables based on type and effect of module signatures, effect systems provide a method to extend abstract interpretation in the context of separate compilation. This separate abstract interpretation makes the control-flow analysis as effective as the abstract interpretation approach on closed expressions, but is also able to tackle expressions with free variables, using their type to approximate their abstract value.

The goal of control-flow analysis is to implement functional languages more efficiently. The static knowledge of control-flow information plays an important role in optimizing compilers of functional languages, such as interprocedural data-flow optimizations and closure optimizations. There remain a lot of open issues in pragmatic use of this control-flow information. We present an application of control-flow information in optimizing *closure allocation*. This optimization is based on *escape analysis*, a direct application of types and control-flow information. The escape analysis identifies the free variables that outlive the lexical scope of function definitions. This compile time knowledge of *escaping variables* helps compilers choose a more efficient allocation strategy for closures, i.e. non-escaping variables can be safely stored in the stack, while heap allocation is only used for escaping ones.

Bibliography

[Aho86] Aho, A. V., Sethi, R. and Ullman, J. D. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[Aiken93] Aiken A. Type Inclusion Constraints and Type Inference. In *Conference on Functional Programming Languages and Computer Architecture*. August, 1993.

[Appel87] Appel, A. W. and MacQueen, D. B. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, volume 242 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[Appel89] Appel, A. W. and Jim, T.Y. Continuation-Passing, Closure-Passing Style. In *ACM Symposium on Principles of Programming Languages*, pages 293-302, 1989.

[Appel90] Appel, A. W. and Mac Queen, D. B. Standard ML Reference Manual. AT&T Bell Laboratories and Princeton University, October 1990.

[Appel92] Appel, A. W. Compiling with Continuations. Cambridge University Press, 1992.

[Baker92] Baker, H.G. CONS Should not CONS its Arguments, or, a Lazy Alloc is Smart Alloc. In *ACM SIGPLAN Notices* Volume 27, No.3, 1992.

[Benjamin92] Benjamin, P. Intersection Types and Bounded Polymorphism. In *LFCS Report Series*, University of Edinburgh, 1992.

[Bondorf93] Bondorf, A. and Jorgensen, J. Efficient Analyses for Realistic Off-line Partial Evaluation. In *Journal of Functional Programming*, Vol 3, Part 3, Cambridge University Press, July 1993.

[Cardelli84] Cardelli, L. Compiling a Functional Language. In *ACM Symposium of LAFP*, 1984.

[Cardelli85] Cardelli, L. On Understanding Types, Data-Abstractions, and Polymorphism. In *Computing Surveys*, 17(4),1985.

[Cardelli88] Cardelli, L. Structural Subtyping and the Notion of Power Type. In *ACM Symposium on Principles of Programming Languages*, pages 70-79, 1988.

[Consel93] Consel, C. and Jouvelot, P. Separate Polyvariant Binding-Time Analysis. In *Technical Report*, CRI, Ecole des Mines de Paris, 1993.

[Cousot77] Cousot, P. and Cousot, R. Abstract Interpretation, a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*. 1977.

[Cousot79] Cousot, P. and Cousot, R. Systematic Design of Program Analysis Frameworks. In *ACM Symposium on Principles of Programming Languages*. 1979.

[Damas82] Damas, L. and Milner, R. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, pages 207-212. 1982.

[Deutsch90] Deutsch A. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-Order Functional Specifications. In *ACM Symposium on Principles of Programming Languages*, 157-168,1990.

[Dornic91] Dornic, V. and Jouvelot, P. Polymorphic Time Systems for Estimating Program Complexity. In *LOPLAS'91, Bordeaux, France*, 1991.

[Fuh88] Fuh. Y. and Mishra, P. Type Inference with Subtypes. In *European Symposium on Programming*, pages 94-114. 1988.

[Gifford87] 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.

[Goldberg90] Goldberg. B. and Park, Y. G. Higher Order Escape Analysis, Optimizing Stack Allocation in Functional Program Implementation. In *European Symposium on Programming*, volume 432 of the *Lectures Notes in Computer Science*, pages 152-160. Springer-Verlag, 1990.

[Grundman92] Grundman., D, Stata, R. and Toole, J.O. Mini-DX/DLX – A Pedagogic Compiler, M.I.T, 1992.

[Hammel88] Hammel, R. T. and Gifford, D. K. FX-87 Performance Measurements: Dataflow Implementation. *MIT/LCS/TR-421*, MIT Laboratory for Computer Science, November 1988.

[Jouvelot88] Jouvelot, P. and Gifford, D. K. The FX-87 Interpreter In *International Conference on Computer Languages*, 1988.

[Jouvelot89] Jouvelot, P. and Gifford, D. K. Reasoning about Continuations with Control Effects. In *International Conference on Progamming Language Design and Implementation*. ACM, New-York, 1989.

[Jouvelot91] Jouvelot, P. and Gifford, D. K. Algebraic Reconstruction of Types and Effects. In *ACM Symposium on Principles of Programming Languages*. 1991.

[Kanellakis89] Kanellakis, P. and Mitchell, J. C. Polymorphic Unification and ML Typing. In *ACM Symposium on Principles of Programming Languages*, 1989.

[Kelsey89] Kelsey, R.A. Compilation by Program Transformation. *Ph.D. Thesis*. Yale University, May 1989.

[Krantz87] Krantz, D. ORBIT: An Optimizing Compiler for Scheme. *Ph.D. Thesis*. Yale University, Feb. 1988.

[Leeuwen90] J. Van Leeuwen. Formal Models and Semantics. In *Handbook of Theoretical Computer Science*, volume B. The MIT press, 1990.

[Leroy90-1] Leroy, X. The ZINC Experiment: an Economical Implementation of the ML Language. Technical report 117, INRIA, 1990.

[Leroy90-2] Leroy, X. Unboxed Objects and Polymorphic Typing In *ACM Symposium on Principles of Programming Languages*, 1990.

[Leroy91] Leroy, X. and Weis, P. Polymorphic Type Inference and Assignment. In *ACM Symposium on Principles of Programming Languages*, 1991.

[Lucassen87] 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.

[Lucassen88] Lucassen, J. M. and Gifford, D. K. Polymorphic Effect Systems. In *ACM Conference on Principles of Programming Languages*. ACM, New-York, 1988.

[MacCracken79] MacCracken, N. Investigation of a Programming Language with a Polymorphic Type Structure. *Ph. D. Thesis*, Syracuse University, 1979.

[MacQueen90] MacQueen, D. B. Modules for Standard ML. In *ACM Conference on Lisp and Functional Programming*, pages 198-207. ACM Press, New-York, 1990.

[Milner78] Milner, R. A Theory for Type Polymorphism in Programming. In *Journal of Computer and Systems Sciences*, Vol. 17, pages 348-375. 1978.

[Milner90] Milner, R., Tofte, M. and Harper, R. The Definition of Standard ML. *The MIT Press*, Cambridge, 1990.

[MIT90] Mini-FX Reference Manual. MIT, 1990.

[Mitchell88] Mitchell, J. C. and Harper, R. The Essence of ML. In *ACM Symposium on Principles of Programming Languages*, 1988.

[Mycroft81] Mycroft, A. Abstract Interpretation and Optimizing Transformations for Applicative Programs. *PhD Thesis*, University of Edinburgh. 1981.

[O'Toole90] 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.

[Plotkin81] Plotkin, G. A Structural Approach to Operational Semantics. *Technical Report DAIMI-FN-19*. Aarhus University, 1981.

[Stansifer88] Stanifer, R. Type Inference with Subtypes. In *ACM Symposium on Principles of Programming Languages*, 1988.

[Robinson65] 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.

[Rees88] Rees, J. and Clinger W., Editors. Fourth Report on the Algorithmic Language Scheme. September 1988.

[Siekmann89] Siekmann, J. H. Unification Theory. In *Journal of Symbolic Computations*, volume 7, pages 207-274. Academic Press, 1989.

[Sheldon90] Sheldon, A. M. and Gifford, D. K. Static Dependent Types for First Class Modules. In *ACM Conference on Lisp and Functional Programming*, 1990.

[Shivers91] 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.

[Steele78] Steele, G. Rabbit: A Compiler for Scheme. In *MIT-AI Technical Report No. 474*. MIT Laboratory for Computer Science, May 1978.

[Steele90] Steele, G. L. *Common Lisp, the language*. Digital Press 1990.

[Stoy77] Stoy, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[Talpin92-1] 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.

[Talpin92-2] Talpin, J. P. and Jouvelot, P. The Type and Effect Discipline. In *IEEE Conference on Logic in Computer Science*. Santa Cruz, California, June 1992.

[Talpin93-1] Talpin, J. P. and Jouvelot, P. Compiling FX on the Connection Machine. In *Workshop on Static Analysis*, Sept 1993.

[Talpin93-2] Talpin, J. P. Type Discipline. *PhD Thesis*. May 1993

[Tang92] 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.

[Tang93] Tang, Y. M. and Jouvelot, P. Effect Systems with Subtyping. In *Technical Report*, CRI, Ecole des Mines de Paris 1993.

[Tang94] Tang, Y. M. and Jouvelot, P. Separate Abstract Interpretation for Control-Flow Analysis. *International Symposium on Theoretical Aspects of Computer Software*. Japan, Avril 1994.

[Tofte87] Tofte, M. Operational Semantics and Polymorphic Type Inference. *PhD Thesis* and Technical Report *ECS-LFCS-88-54*, University of Edinburgh, 1987.

[Tofte90] Tofte, M. Type Inference for Polymorphic References. In *Information and Computation*, 89(1), pages 1-34, 1990.

[Wand87] Wand, M. A simple algorithm and proof for type inference. In *Fundamenta Informaticae*, volume 10, pages 115-122. North Holland, 1987.

[Wand93] Wand, M. and Steckler, P. Selective and Lightweight Closure Conversion In *Technical Report* 1993

Appendix 1

Proof of Lemma 3.1

Lemma 3.1 [Monotony of $\mathcal{F}$] If $\mathcal{Q}$ and $\mathcal{Q}'$ are two subsets of the domain $\mathcal{R}$.

$$\mathcal{Q} \subseteq \mathcal{Q}' \Rightarrow \mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$$

Proof

Taking any $q = (v, t)$, *s.t.* $q \in \mathcal{F}(\mathcal{Q})$

Since $q \in \mathcal{F}(\mathcal{Q})$, by the definition of $\mathcal{F}$
(1) $q \in \mathcal{Q}$

Since $\mathcal{Q} \subseteq \mathcal{Q}'$
(2) $q \in \mathcal{Q}'$

Now we prove that $q \in \mathcal{F}(\mathcal{Q}')$

- Case $v = i$

  Since $q \in \mathcal{F}(\mathcal{Q})$, by the definition of $\mathcal{F}$
  (3) $q = (i, int)$

  From (2)(3), by the definition of $\mathcal{F}$
  (4) $q \in \mathcal{F}(\mathcal{Q}')$

- Case $v = (\mathtt{n}, \mathtt{x}, \mathtt{e}, E)$

  Since $q \in \mathcal{F}(\mathcal{Q})$, by the definition of $\mathcal{F}$, $\exists \mathcal{E}$ *s.t.*
  (5) $\forall \mathtt{x} \in Dom(E),\ \mathtt{x} \in Dom(\mathcal{E})$
  (6) $(E(\mathtt{x}), \mathcal{E}(\mathtt{x})) \in \mathcal{Q}$
  (7) $\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t$

  From (6), since $\mathcal{Q} \subseteq \mathcal{Q}'$
  (8) $(E(\mathtt{x}), \mathcal{E}(\mathtt{x})) \in \mathcal{Q}'$

  From (2)(5)(8)(7), by the definition of $\mathcal{F}$
  (9) $q \in \mathcal{F}(\mathcal{Q}')$

From (4)(9)
(10) $\forall q \in \mathcal{F}(\mathcal{Q}), q \in \mathcal{F}(\mathcal{Q}')$

From (10)
$\mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$ ♣

Proof of Theorem 3.1

Theorem 3.1 [Consistency of Static Semantics]

$$\left.\begin{array}{l} E \vdash \mathtt{e} \rightarrow v, b \\ \mathcal{E} \vdash \mathtt{e} : t, c \\ E : \mathcal{E} \end{array}\right\} \Rightarrow \left\{\begin{array}{l} v : t \\ b \subseteq c \end{array}\right.$$

Proof By induction on the number of reduction steps of expressions.

- Case of (*var*)

  The hypotheses are
  (1) $E : \mathcal{E}$
  (2) $E \vdash \mathtt{x} \rightarrow E(\mathtt{x}), \emptyset$
  (3) $\mathcal{E} \vdash \mathtt{x} : \mathcal{E}(\mathtt{x}), \emptyset$

  From (2), by (*var*) in the dynamic semantics
  (4) $\mathtt{x} \in Dom(E)$

  From (3), by (*var*) in the static semantics
  (5) $\mathtt{x} \in Dom(\mathcal{E})$

  From (1)(4)(5), by Definition 3.1
  $E(\mathtt{x}) : \mathcal{E}(\mathtt{x})$

- Case of (*abs*)

  The hypotheses are
  (1) $E : \mathcal{E}$
  (2) $E \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) \rightarrow (\mathtt{n}, \mathtt{x}, \mathtt{e}, E_{\mathtt{x}}), \emptyset$
  (3) $\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t' \stackrel{\{\mathtt{n}\}\cup c}{\rightarrow} t, \emptyset$

  From (1), by Definition 3.1
  (4) $E_{\mathtt{x}} : \mathcal{E}$

  From (3)(4), by Definition 3.1
  $(\mathtt{n}, \mathtt{x}, \mathtt{e}, E_{\mathtt{x}}) : t' \stackrel{\{\mathtt{n}\}\cup c}{\rightarrow} t$

- Case of (*rec*)

  The hypotheses are
  (1) $E : \mathcal{E}$
  (2) $E \vdash (\mathtt{rec_n}\ (\mathtt{f}\ \mathtt{x})\ \mathtt{e}) \rightarrow cl, \emptyset$
  (3) $\mathcal{E} \vdash (\mathtt{rec_n}\ (\mathtt{f}\ \mathtt{x})\ \mathtt{e}) : t, \emptyset$

  From (2), by (*rec*) in the dynamic semantics
  (4) $cl = (\mathtt{n}, \mathtt{x}, \mathtt{e}, E')$
  where $E' = E[\mathtt{f} \mapsto cl]$

  From (3), by (*rec*) in the static semantics
  (5) $\mathcal{E}' \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t, \emptyset$
  where $\mathcal{E}' = \mathcal{E}[\mathtt{f} \mapsto t]$

  By Definition 3.2, for proving $cl : t$
  we have to prove : $(cl, t) \in gfp(\mathcal{F})$

  We define $\mathcal{Q} = gfp(\mathcal{F}) \cup \{(cl, t)\}$

  By the definition of $gfp(\mathcal{F})$, for proving $(cl, t) \in gfp(\mathcal{F})$
  we have to prove : $Q \subseteq \mathcal{F}(\mathcal{Q})$

  Taking any q, *s.t.* $q \in \mathcal{Q}$

  - Case $q \in gfp(\mathcal{F})$
    Since $gfp(\mathcal{F}) \subseteq \mathcal{Q}$ and $\mathcal{F}$ is monotonic
    (6) $\mathcal{F}(gfp(\mathcal{F})) \subseteq \mathcal{F}(\mathcal{Q})$

    By the definition of $gfp(\mathcal{F})$
    (7) $gfp(\mathcal{F}) \subseteq \mathcal{F}(gfp(\mathcal{F}))$

    From (6)(7)
    (8) $gfp(\mathcal{F}) \subseteq \mathcal{F}(\mathcal{Q})$
    thus $q \in \mathcal{F}(\mathcal{Q})$

  - Case $q = (cl, t)$
    From (1), by Definition 3.2
    $\forall \mathtt{x} \in Dom(E), \mathtt{x} \in Dom(\mathcal{E})$
    (9) $(E(\mathtt{x}), \mathcal{E}(\mathtt{x})) \in gfp(\mathcal{F})$

    From (9), since $gfp(\mathcal{F}) \subseteq \mathcal{Q}$
    (10) $(E(\mathtt{x}), \mathcal{E}(\mathtt{x})) \in \mathcal{Q}$

From (10), since $(cl, t) \in \mathcal{Q}$
$\forall \mathtt{x} \in Dom(E)$, $\mathtt{x} \in Dom(\mathcal{E})$
(11) $(E'(\mathtt{x}), \mathcal{E}'(\mathtt{x})) \in \mathcal{Q}$

From (11)(5), since $(cl, t) \in \mathcal{Q}$, by the definition of $\mathcal{F}$
(12) $(cl, t) \in \mathcal{F}(\mathcal{Q})$
thus $q \in \mathcal{F}(\mathcal{Q})$

From (8)(12)
$\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$

By the definition of $gfp(\mathcal{F})$, since $(cl, t) \in \mathcal{Q}$
$(cl, t) \in gfp(\mathcal{F})$

By Definition 3.2
$cl : t$

- Case of (app)

  The hypotheses are
  (1) $E : \mathcal{E}$
  (2) $E \vdash (\mathtt{e}\ \mathtt{e}') \rightarrow v, b \cup b' \cup b'' \cup \{\mathtt{n}\}$
  (3) $\mathcal{E} \vdash (\mathtt{e}\ \mathtt{e}') : t, c \cup c' \cup c''$

  From (2), by (app) in the dynamic semantics
  (4) $E \vdash \mathtt{e} \rightarrow (\mathtt{n}, \mathtt{x}, \mathtt{e}'', E'), b$
  (5) $E \vdash \mathtt{e}' \rightarrow v', b'$
  (6) $E'[\mathtt{x} \mapsto v'] \vdash \mathtt{e}'' \rightarrow v, b''$

  From (3), by (app) in the static semantics
  (7) $\mathcal{E} \vdash \mathtt{e} : t' \xrightarrow{c''} t, c$
  (8) $\mathcal{E} \vdash \mathtt{e}' : t', c'$

  From (1)(4)(7) and (1)(5)(8), by inductions
  (9) $(\mathtt{n}, \mathtt{x}, \mathtt{e}'', E') : t' \xrightarrow{c''} t$
  (10) $b \subseteq c$

  (11) $v' : t'$
  (12) $b' \subseteq c'$

  From (9), by Definition 3.1
  (13) $\exists \mathcal{E}'$, verifying $E' : \mathcal{E}'$

(14) $\mathcal{E}' \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}'') : t' \stackrel{c''}{\rightarrow} t$

From (14), by (*abs*) in the static semantics
(15) $\mathcal{E}'[\mathtt{x} \mapsto t'] \vdash \mathtt{e}'' : t, c'''$
(16) $c'' = c''' \cup \{\mathtt{n}\}$

From (13)(11)
(17) $E'[\mathtt{x} \mapsto v'] : \mathcal{E}'[\mathtt{x} \mapsto t']$

From (17)(6)(15), by induction
$v : t$
(18) $b'' \subseteq c'''$

From (10)(12)(18)(16)
$b \cup b' \cup b'' \cup \{\mathtt{n}\} \subseteq c \cup c' \cup c''$

Appendix 2

Proof of Theorem 4.4

Theorem 4.4 [Soundness] Given an expression $\mathtt{e}$ and its type environment $\mathcal{E}$, if $\mathcal{R}(\mathcal{E}, \mathtt{e}) = \langle \theta, t, c, \kappa \rangle$, then, for any effect model μ of κ, one has:

$$\mu\theta\mathcal{E} \vdash \mathtt{e} : \mu t, \mu c$$

Proof By induction on the number of reduction steps of expressions.

- Case of (var)

  The hypotheses are
  (1) $\mathcal{R}(\mathcal{E}, \mathtt{x}) = \langle Id, \mathcal{E}(\mathtt{x}), \emptyset, \emptyset \rangle$
  (2) $\mu \models \emptyset$

  From (1), by the definition of $\mathcal{R}$
  (3) $\mathtt{x} \in Dom(\mathcal{E})$, i.e. $\mathtt{x} \in Dom(\mu\mathcal{E}(\mathtt{x}))$

  From (3), by (var) in the static semantics
  $\mu\mathcal{E} \vdash \mathtt{x} : \mu\mathcal{E}(\mathtt{x}), \emptyset$

- Case of (abs)

  The hypotheses are
  (1) $\mathcal{R}(\mathcal{E}, (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e})) = \langle \theta, \theta\alpha \xrightarrow{\zeta} t, \emptyset, \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\} \rangle$
  (2) $\mu \models \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$

  From (1), by the definition of $\mathcal{R}$
  (3) $\mathcal{R}(\mathcal{E}[\mathtt{x} \mapsto \alpha], \mathtt{e}) = \langle \theta, t, c, \kappa \rangle$

  From (2), by the definition of effect models
  (4) $\mu \models \kappa$
  (5) $\mu \models \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$, i.e. $\mu\zeta \supseteq \{\mathtt{n}\} \cup \mu c$

  From (3)(4), by induction
  (6) $\mu\theta(\mathcal{E}[\mathtt{x} \mapsto \alpha]) \vdash \mathtt{e} : \mu t, \mu c$

From (6)(5), by (*does*) in the static semantics
(7) $\mu\theta\mathcal{E}[\mathtt{x} \mapsto \mu\theta\alpha]) \vdash \mathtt{e} : \mu t, \mu\zeta$

From (7), by (*abs*) in the static semantics
$\mu\theta\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : \mu(\theta\alpha \xrightarrow{\zeta} t), \emptyset$

- Case of (*rec*)

  The hypotheses are
  (1) $\mathcal{R}(\mathcal{E}, (\mathtt{rec}_{\mathtt{n}}\ (\mathtt{f}\ \mathtt{x})\ \mathtt{e})) = \langle \theta'\theta, \theta'\theta(\alpha' \xrightarrow{\zeta} \alpha), \emptyset, \theta'(\kappa \cup \{\theta\zeta \supseteq \{\mathtt{n}\} \cup c\})\rangle$
  (2) $\mu \models \theta'(\kappa \cup \{\theta\zeta \supseteq \{\mathtt{n}\} \cup c\})$

  From (1), by the definition of $\mathcal{R}$
  (3) $\mathcal{R}(\mathcal{E}[\mathtt{f} \mapsto \alpha' \xrightarrow{\zeta} \alpha][\mathtt{x} \mapsto \alpha'], \mathtt{e}) = \langle \theta, t, c, \kappa\rangle$
  (4) $\theta' = \mathcal{U}(\theta\alpha, t)$
  where α', α, ζ are fresh

  From (2), by the definition of effect models
  (5) $\mu\theta' \models \kappa$
  (6) $\mu\theta' \models \theta\zeta \supseteq \{\mathtt{n}\} \cup c$, i.e. $\mu\theta'\theta\zeta \supseteq \{\mathtt{n}\} \cup \mu\theta' c$

  From (3)(5), by induction
  (7) $\mu\theta'\theta(\mathcal{E}[\mathtt{f} \mapsto \alpha' \xrightarrow{\zeta} \alpha][\mathtt{x} \mapsto \alpha']) \vdash \mathtt{e} : \mu\theta' t, \mu\theta' c$

  From (7)(6), by (*does*) rule in the static semantics
  (8) $\mu\theta'\theta(\mathcal{E}[\mathtt{f} \mapsto \alpha' \xrightarrow{\zeta} \alpha][\mathtt{x} \mapsto \alpha']) \vdash \mathtt{e} : \mu\theta' t, \mu\theta'\theta\zeta$

  From (4), by the correctness of unification
  (9) $\theta'\theta\alpha = \theta' t$ i.e $\mu\theta'\theta\alpha = \mu\theta' t$

  From (8)(9)
  (10) $\mu\theta'\theta(\mathcal{E}[\mathtt{f} \mapsto \alpha' \xrightarrow{\zeta} \alpha][\mathtt{x} \mapsto \alpha']) \vdash \mathtt{e} : \mu\theta'\theta\alpha, \mu\theta'\theta\zeta$

  From (10), by (*rec*) in the static semantics
  $\mu\theta'\theta\mathcal{E} \vdash (\mathtt{rec}_{\mathtt{n}}\ (\mathtt{f}\ \mathtt{x})\ \mathtt{e}) : \mu\theta'\theta(\alpha' \xrightarrow{\zeta} \alpha), \emptyset$

- Case of (*app*)

  The hypotheses are
  (1) $\mathcal{R}(\mathcal{E}, (\mathtt{e}\ \mathtt{e}')) = \langle \theta''\theta'\theta, \theta''\alpha, \theta''(\theta' c \cup c' \cup \zeta), \theta''(\theta'\kappa \cup \kappa')\rangle$
  (2) $\mu \models \theta''(\theta'\kappa \cup \kappa')$

From (1), by the definition of $\mathcal{R}$
(3) $\mathcal{R}(\mathcal{E}, \mathtt{e}) = \langle \theta, t, c, \kappa \rangle$
(4) $\mathcal{R}(\theta\mathcal{E}, \mathtt{e}') = \langle \theta', t', c', \kappa' \rangle$
(5) $\theta'' = \mathcal{U}(\theta' t, t' \xrightarrow{\zeta} \alpha)$
for fresh variables α and ζ

From (2), by the definition of effect models
(6) $\mu\theta''\theta' \models \kappa$
(7) $\mu\theta'' \models \kappa'$

From (3)(6) and (4)(7), by inductions
(8) $\mu\theta''\theta'\theta\mathcal{E} \vdash \mathtt{e} : \mu\theta''\theta' t, \mu\theta''\theta' c$
(9) $\mu\theta''\theta'\theta\mathcal{E} \vdash \mathtt{e}' : \mu\theta'' t', \mu\theta'' c'$

From (5), by the correctness of unification
(10) $\theta''\theta' t = \theta''(t' \xrightarrow{\zeta} \alpha)$, i.e. $\mu\theta''\theta' t = \mu\theta'' t' \xrightarrow{\mu\theta''\zeta} \mu\theta''\alpha$

From (8)(9)(10), by (*app*) in the static semantics
$\mu\theta''\theta'\theta\mathcal{E} \vdash (\mathtt{e}\ \mathtt{e}') : \mu\theta''\alpha, \mu\theta''(\theta' c \cup c' \cup \zeta)$ ♣

Proof of Theorem 4.5

Theorem 4.5 [Completeness] If $\theta_1\mathcal{E} \vdash \mathtt{e} : t_1, c_1$, then $\mathcal{R}(\mathcal{E}, \mathtt{e}) = \langle \theta, t, c, \kappa \rangle$ and there exists an effect model μ of κ such that:

$$\theta_1\mathcal{E} = \mu\theta\mathcal{E} \quad \textit{and} \quad t_1 = \mu t \quad \textit{and} \quad c_1 \supseteq \mu c$$

Proof By induction on the number of reduction steps of expressions.

- Case of (*var*)

  The hypothesis is
  $\theta_1\mathcal{E} \vdash \mathtt{x} : \theta_1\mathcal{E}(\mathtt{x}), \emptyset$

  By (*var*) in the static semantics
  $\mathtt{x} \in \mathit{Dom}(\mathcal{E})$

  By the definition of $\mathcal{R}$
  $\mathcal{R}(\mathcal{E}, \mathtt{x}) = \langle \mathit{Id}, \mathcal{E}(\mathtt{x}), \emptyset, \emptyset \rangle$

  Taking $\mu = \theta_1$, such that $\mu \models \emptyset$
  $\theta_1\mathcal{E} = \mu\mathcal{E}$
  $\theta_1\mathcal{E}(\mathtt{x}) = \mu\mathcal{E}(\mathtt{x})$

- Case of (abs)

  The hypothesis is
  $\theta_1\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t_1' \xrightarrow{\{\mathtt{n}\}\cup c_1} t_1, \emptyset$

  By (abs) in the static semantics
  (1) $\theta_1\mathcal{E}[\mathtt{x} \mapsto t_1'] \vdash \mathtt{e} : t_1, c_1$

  Suppose α new, we define a substitution θ_1', such that :

$$\theta_1'\vartheta = \begin{cases} t_1' & \vartheta = \alpha \\ \theta_1\vartheta & otherwise \end{cases}$$

  By the definition of θ_1', (1) is equivalent to
  (2) $\theta_1'(\mathcal{E}[\mathtt{x} \mapsto \alpha]) \vdash \mathtt{e} : t_1, c_1$

  From (2), by induction
  (3) $\mathcal{R}(\mathcal{E}[\mathtt{x} \mapsto \alpha], \mathtt{e}) = \langle\theta, t, c, \kappa\rangle$
  $\exists\mu$, $\mu \models \kappa$, such that
  (4) $\theta_1'(\mathcal{E}[\mathtt{x} \mapsto \alpha]) = \mu\theta(\mathcal{E}[\mathtt{x} \mapsto \alpha])$
  (5) $t_1 = \mu t$
  (6) $c_1 \supseteq \mu c$

  From (3), by the definition of $\mathcal{R}$
  $\mathcal{R}(\mathcal{E}, (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e})) = \langle\theta, \theta\alpha \xrightarrow{\zeta} t, \emptyset, \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}\rangle$
  where ζ is fresh

  We define a substitution μ' on $fv(\theta\mathcal{E}, \theta\alpha, t, c, \kappa)$ and ζ.

$$\mu'\vartheta = \begin{cases} \mu(\vartheta) & \vartheta \in fv(\theta\mathcal{E}, \theta\alpha, t, c, \kappa) \\ \{\mathtt{n}\} \cup c_1 & \vartheta = \zeta \end{cases}$$

  By the definition of μ'
  (7) $\mu'\zeta = \{\mathtt{n}\} \cup c_1$
  (8) $\mu'(\{\mathtt{n}\} \cup c) = \{\mathtt{n}\} \cup \mu c$
  (9) $\mu'\kappa = \mu\kappa$

  From (6)(7)(8), by the definition of effect models
  (10) $\mu' \models \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$

  From (9), since $\mu \models \kappa$
  (11) $\mu' \models \kappa$

From (10)(11), by the definition of effect models
$\mu' \models \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$

From (4), by the definition of θ_1' and μ'
$\theta_1 \mathcal{E} = \mu\theta\mathcal{E} = \mu'\theta\mathcal{E}$
(12) $t_1' = \mu\theta\alpha = \mu'\theta\alpha$

From (5), by the definition of μ'
(13) $t_1 = \mu' t$

From (7)(12)(13)
$t_1' \overset{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1 = \mu'(\theta\alpha \overset{\zeta}{\rightarrow} t)$

- Case of (*rec*)

  The hypothesis is
  $\theta_1\mathcal{E} \vdash (\mathtt{rec_n\ (f\ x)\ e}) : t_1' \overset{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1, \emptyset$

  By (*rec*) in the static semantics
  (1) $\theta_1\mathcal{E}[\mathtt{f} \mapsto t_1' \overset{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1][\mathtt{x} \mapsto t_1'] \vdash \mathtt{e} : t_1, c_1$

  Suppose α', α, ζ new, we define a substitution θ_1', such that :

$$\theta_1'\vartheta = \begin{cases} t_1' & \vartheta = \alpha' \\ t_1 & \vartheta = \alpha \\ \{\mathtt{n}\} \cup c_1 & \vartheta = \zeta \\ \theta_1\vartheta & \textit{otherwise} \end{cases}$$

  By the definition of θ_1', (1) is equivalent to
  (2) $\theta_1'(\mathcal{E}[\mathtt{f} \mapsto \alpha' \overset{\zeta}{\rightarrow} \alpha][\mathtt{x} \mapsto \alpha']) \vdash \mathtt{e} : t_1, c_1$

  From (2), by induction
  (3) $\mathcal{R}(\mathcal{E}[\mathtt{f} \mapsto \alpha' \overset{\zeta}{\rightarrow} \alpha][\mathtt{x} \mapsto \alpha'], \mathtt{e}) = \langle\theta, t, c, \kappa\rangle$
  $\exists\mu$, $\mu \models \kappa$, such that
  (4) $\theta_1'(\mathcal{E}[\mathtt{f} \mapsto \alpha' \overset{\zeta}{\rightarrow} \alpha][\mathtt{x} \mapsto \alpha']) = \mu\theta(\mathcal{E}[\mathtt{f} \mapsto \alpha' \overset{\zeta}{\rightarrow} \alpha][\mathtt{x} \mapsto \alpha'])$

  (5) $t_1 = \mu t$
  (6) $c_1 \supseteq \mu c$

  From (4), By the definition of θ_1'
  (7) $\theta_1\mathcal{E} = \mu\theta\mathcal{E}$
  (8) $t_1' \overset{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1 = \mu\theta(\alpha' \overset{\zeta}{\rightarrow} \alpha)$
  (9) $t_1 = \mu\theta\alpha$

(10) $\{\mathtt{n}\} \cup c_1 = \mu\theta\zeta$

From (5)(9)
(11) $\mu t = \mu\theta\alpha$

From (11), by the correctness of unification
(12) $\exists\theta', \theta' = \mathcal{U}\ (t, \theta\alpha)$
and $\exists\mu'$, such that:
(13) $\mu = \mu'\theta'$

From (3)(12), by the definition of $\mathcal{R}$
$\mathcal{R}(\mathcal{E}, (\mathtt{rec_n}\ (\mathtt{n}, \mathtt{x})\ \mathtt{e})) = \langle \theta'\theta, \theta'\theta(\alpha' \stackrel{\{\mathtt{n}\}\cup c}{\rightarrow} \alpha), \emptyset, \theta'(\kappa \cup \{\theta\zeta \supseteq \{\mathtt{n}\} \cup c\})\rangle$

From (13), since $\mu \models \kappa$
(14) $\mu' \models \theta'\kappa$

From (6)(10)
(15) $\mu\theta\zeta \supseteq \{\mathtt{n}\} \cup \mu c$

From (15)(13), by the definition of effect models
(16) $\mu' \models \{\theta'\theta\zeta \supseteq \{\mathtt{n}\} \cup \theta' c\}$

From (14)(16), by the definition of the effect models
$\mu' \models \theta'(\kappa \cup \{\theta\zeta \supseteq \{\mathtt{n}\} \cup c\})$

From (7)(13)
$\theta_1\mathcal{E} = \mu\theta\mathcal{E} = \mu'\theta'\theta\mathcal{E}$

From (8)(13)
$t_1' \stackrel{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1 = \mu'\theta'\theta(\alpha' \stackrel{\zeta}{\rightarrow} \alpha)$

- Case of (*app*)

  The hypothesis is
  $\theta_1\mathcal{E} \vdash (\mathtt{e}\ \mathtt{e}') : t_1, c_1 \cup c_1' \cup c_1''$

  By (*app*) in the static semantics
  (1) $\theta_1\mathcal{E} \vdash \mathtt{e} : t_1' \stackrel{c_1''}{\rightarrow} t_1, c_1$
  (2) $\theta_1\mathcal{E} \vdash \mathtt{e}' : t_1', c_1'$

  From (1), by induction
  (3) $\mathcal{R}(\mathcal{E}, \mathtt{e}) = \langle \theta, t, c, \kappa\rangle$

$\exists \mu$, $\mu \models \kappa$, such that:
(4) $\theta_1 \mathcal{E} = \mu\theta\mathcal{E}$
(5) $t_1' \xrightarrow{c_1''} t_1 = \mu t$
(6) $c_1 \supseteq \mu c$

From (4), (2) is equivalent with:
(7) $\mu(\theta\mathcal{E}) \vdash \mathsf{e}' : t_1', c_1'$

From (7), by induction
(8) $\mathcal{R}(\theta\mathcal{E}, \mathsf{e}') = \langle \theta', t', c', \kappa' \rangle$
$\exists \mu'$, $\mu' \models \kappa'$, such that:
(9) $\mu\theta\mathcal{E} = \mu'\theta'\theta\mathcal{E}$
(10) $t_1' = \mu' t'$
(11) $c_1' \supseteq \mu' c'$

Let α, ζ be fresh. We define a substitution θ_0 on $fv(\theta\mathcal{E}, t, c, \kappa)$, $fv(\theta'\theta\mathcal{E}, t', c', \kappa')$, α and ζ

$$\theta_0\vartheta = \begin{cases} \mu\vartheta & \vartheta \in fv(\theta\mathcal{E}, t, c, \kappa) \\ \mu'\vartheta & \vartheta \in fv(\theta'\theta\mathcal{E}, t', c', \kappa') \\ t_1 & \vartheta = \alpha \\ c_1'' & \vartheta = \zeta \end{cases}$$

Note that $\forall \vartheta$, if $\vartheta \in fv(\theta\mathcal{E}, t, c, \kappa)$ and $\vartheta \in fv(\theta'\theta\mathcal{E}, t', c', \kappa')$
then by the definition of $\mathcal{R}$, $\vartheta \in fv(\theta\mathcal{E})$ and $\vartheta \in fv(\theta'\theta\mathcal{E})$, which means $\theta'\vartheta = \vartheta$
From (9), we know $\mu\vartheta = \mu'\vartheta$, thus θ_0 is well defined.

From (10), by the definition of θ_0,
(12) $\theta_0(t' \xrightarrow{\zeta} \alpha) = \mu' t' \xrightarrow{c_1''} t_1 = t_1' \xrightarrow{c_1''} t_1$

By the definition of $\mathcal{R}$, $\forall \vartheta \in fv(t, c, \kappa)$
v is either in $fv(\theta\mathcal{E})$ or is a fresh variable introduced by $\mathcal{R}(\mathcal{E}, \mathsf{e})$.

- case $\vartheta \in fv(\theta\mathcal{E})$
  By the definition of θ_0
  (13) $\theta_0(\theta'\theta\mathcal{E}) = \mu'(\theta'\theta\mathcal{E})$

  From (9)(13)
  (14) $\theta_0\theta'(\theta\mathcal{E}) = \mu(\theta\mathcal{E})$, i. e $\theta_0\theta'\vartheta = \mu\vartheta$

- case ϑ is fresh
  Since $\vartheta \in fv(t, c, \kappa)$, by the definition of θ_0
  (15) $\theta_0\vartheta = \mu\vartheta$

From (15), since $\theta'\vartheta = \vartheta$
(16) $\theta_0\theta'\vartheta = \theta_0\vartheta = \mu\vartheta$

From (14)(16)
(17) $\forall\vartheta \in fv(t, c, \kappa)$, $\theta_0\theta' = \mu$

From (5)(17)
(18) $t_1' \stackrel{c_1''}{\rightarrow} t_1 = \theta_0\theta' t$

From (12)(18)
(19) $\theta_0(\theta' t) = \theta_0(t' \stackrel{\zeta}{\rightarrow} \alpha)$

From (19), by the correctness of unification, $\exists\theta''$ such that
(20) $\theta'' = \mathcal{U}(\theta' t, t' \stackrel{\zeta}{\rightarrow} \alpha)$ and $\exists\mu''$ such that
(21) $\theta_0 = \mu''\theta''$

From (3)(8)(20), by the definition of $\mathcal{R}$
$\mathcal{R}(\mathcal{E}, (\mathtt{e}\ \mathtt{e}')) = \langle \theta''\theta'\theta, \theta''\alpha, \theta''(\theta' c \cup c' \cup \zeta), \theta''(\theta'\kappa \cup \kappa') \rangle$

From (17), since $\mu \models \kappa$
(22) $\theta_0\theta' \models \kappa$

Since $\mu' \models \kappa'$, by the definition of θ_0
(23) $\theta_0 \models \kappa'$

From (22)(23), by the definition of effect models
(24) $\theta_0 \models \theta'\kappa \cup \kappa'$

From (24)(21), by the definition of effect models
$\mu'' \models \theta''(\theta'\kappa \cup \kappa')$

From (4)(9)(21), by the definition of θ_0
(25) $\theta_1\mathcal{E} = \mu'(\theta'\theta\mathcal{E}) = \theta_0(\theta'\theta\mathcal{E}) = \mu''\theta''\theta'\theta\mathcal{E}$

From (21), since $\theta_0\alpha = t_1$
$t_1 = \mu''\theta''\alpha$

From (6)(17)(21)
(26) $c_1 \supseteq \mu c = \theta_0\theta' c = \mu''\theta''\theta' c$

From (11)(21), by the definition of θ_0
(27) $c_1' \supseteq \mu' c' = \theta_0 c' = \mu'' \theta'' c'$

From (21), by the definition of θ_0
(28) $c_1'' = \theta_0 \zeta = \mu'' \theta'' \zeta$

From (26)(27)(28)
$c_1 \cup c_1' \cup c_1'' \supseteq \mu'' \theta'' (\theta' c \cup c' \cup \zeta)$ ♣

Appendix 3

Proof of Lemma 5.2

Lemma 5.2 [Formal Effect Constraints] If $\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t, c, \kappa \rangle$, then κ is of form:

$$\{\zeta_i \supseteq c_i \mid i = 1..s\}$$

Proof By induction of the structure of expressions.

- Case of $\mathtt{x}$

  The hypothesis is
  $\mathcal{S}(\mathcal{E}, \mathtt{x}) = \langle t, \emptyset, \mathit{Eff}(t' \leq t) \rangle$

  By the definition of $\mathcal{S}$
  (1) $t' = \mathcal{E}(\mathtt{x})$
  (2) $t = \mathit{New}(\mathit{Erase}(t'))$

  From (1), by Lemma 5.1
  (3) t' includes only fresh effect variables

  From (2)(3), by the definition of *Eff*
  $\mathit{Eff}(t' \leq t)$ verifies the lemma.

- Case of $(\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e})$

  The hypothesis is
  $\mathcal{S}(\mathcal{E}, (\lambda_{\mathtt{n}}\ (\mathtt{x} : \tau)\ \mathtt{e})) = \langle t' \xrightarrow{\zeta} t, \emptyset, \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\} \rangle$

  By the definition of $\mathcal{S}$
  (1) ζ *new*
  (2) $\langle t, c, \kappa \rangle = \mathcal{S}(\mathcal{E}[\mathtt{x} \mapsto t'], \mathtt{e})$

  From (2), by induction
  (3) κ verifies the lemma

  From (1)(3)
  $\kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$ verifies the lemma

- Case of $(\texttt{rec}_\texttt{n}\ (\texttt{f}\ \texttt{x})\ \texttt{e})$

  The hypothesis is
  $\mathcal{S}(\mathcal{E}, (\texttt{rec}_\texttt{n}\ (\texttt{f} : \tau' \to \tau\ \texttt{x} : \tau')\ \texttt{e})) = \langle t' \xrightarrow{\zeta} t, \emptyset, \kappa \cup \mathit{Eff}(t'' \leq t) \cup \{\zeta \supseteq \{\texttt{n}\} \cup c\}\rangle$

  By the definition of $\mathcal{S}$
  (1) $t' \xrightarrow{\zeta} t = \mathit{New}(\tau' \to \tau)$
  (2) $\langle t'', c, \kappa\rangle = \mathcal{S}(\mathcal{E}[\texttt{f} \mapsto t' \xrightarrow{\zeta} t][\texttt{x} \mapsto t'], \texttt{e})$

  From (1), by the definition of New
  (3) $t = \mathit{New}(\tau)$
  (4) ζ *new*

  From (2), by Lemma 5.1
  (5) t'' includes only fresh effect variables

  From (2), by induction
  (6) κ verifies the lemma

  From (3)(5), by the definition of *Eff*
  (7) $\mathit{Eff}(t'' \leq t)$ verifies the lemma

  From (6)(7)(4)
  $\kappa \cup \mathit{Eff}(t'' \leq t) \cup \{\zeta \supseteq \{\texttt{n}\} \cup c\}$ verifies the lemma

- Case of $(\texttt{e}\ \texttt{e}')$

  The hypothesis is
  $\mathcal{S}(\mathcal{E}, (\texttt{e}\ \texttt{e}')) = \langle t, c \cup c' \cup \zeta, \kappa \cup \kappa' \cup \mathit{Eff}(t' \leq t'')\rangle$

  By the definition of $\mathcal{S}$
  (1) $\langle t'' \xrightarrow{c''} t, c, \kappa\rangle = \mathcal{S}(\mathcal{E}, \texttt{e})$
  (2) $\langle t', c', \kappa'\rangle = \mathcal{S}(\mathcal{E}, \texttt{e}')$

  From (1)(2), by Lemma 5.1
  (3) $t'' \xrightarrow{c''} t$ includes only fresh effect variables
  (4) t' includes only fresh effect variables

  From (3)
  (5) t'' includes only fresh effect variables

  From (1)(2), by induction
  (6) κ and κ' verify the lemma

From (6)(4)(5), by the definition of *Eff*
$\kappa \cup \kappa' \cup Eff(t' \leq t'')$ verifies the lemma ♣

Proof of Theorem 5.1

Theorem 5.1 [Soundness] Given an expression $\mathtt{e}$ and its type environment $\mathcal{E}$, if $\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t, c, \kappa \rangle$, then for any effect model μ of κ, one has :

$$\mu\mathcal{E} \vdash \mathtt{e} : \mu t, \mu c$$

Proof By induction on the structure of expressions

- Case of (*var*)

  The hypotheses are
  (1) $\mathcal{S}(\mathcal{E}, \mathtt{x}) = \langle t, \emptyset, Eff(t' \leq t) \rangle$
  (2) $\mu \models \{t' \leq t\}$

  From (1), by the definition of $\mathcal{S}$
  (3) $t' = \mathcal{E}(\mathtt{x})$, i.e. $\mu\mathcal{E}(\mathtt{x}) = \mu t'$

  From (3), by (*var*) rule in the static semantics
  (4) $\mu\mathcal{E} \vdash \mathtt{x} : \mu t', \emptyset$

  From (2), by Lemma 5.3
  (5) $\mu t' \leq \mu t$

  From (4)(5), by (*sub*) rule in the static semantics
  $\mu\mathcal{E} \vdash \mathtt{x} : \mu t, \emptyset$

- Case of (*abs*)

  The hypotheses are
  (1) $\mathcal{S}(\mathcal{E}, (\lambda_{\mathtt{n}}\ (\mathtt{x} : \tau)\ \mathtt{e})) = \langle t' \xrightarrow{\zeta} t, \emptyset, \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\} \rangle$
  (2) $\mu \models \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$
  where $t' = New(\tau)$ and ζ *new*

  From (1), by the definition of $\mathcal{S}$
  (3) $\langle t, c, \kappa \rangle = \mathcal{S}(\mathcal{E}[\mathtt{x} \mapsto t'], \mathtt{e})$

  From (2), by the definition of effect models
  (4) $\mu \models \kappa$
  (5) $\mu \models \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$ i.e. $\mu\zeta \supseteq \mu(\{\mathtt{n}\} \cup c)$

From (3)(4), by induction
(6) $\mu(\mathcal{E}[\mathtt{x} \mapsto t']) \vdash \mathtt{e} : \mu t, \mu c$

From (6), by (*abs*) in the static semantics
(7) $\mu\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : \mu(t' \xrightarrow{\{\mathtt{n}\}\cup c} t), \emptyset$

From (5), by the definition of subtype relation
(8) $\mu(t' \xrightarrow{\{\mathtt{n}\}\cup c} t) \leq \mu(t' \xrightarrow{\zeta} t)$

From (7)(8), by (*sub*) rule in the static semantics
$\mu\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : \mu(t' \xrightarrow{\zeta} t), \emptyset$

- Case of (*rec*)

  The hypotheses are
  (1) $\mathcal{S}(\mathcal{E}, (\mathtt{rec}_{\mathtt{n}}\ (\mathtt{f} : \tau' \to \tau\ \mathtt{x} : \tau')\ \mathtt{e})) = \langle t' \xrightarrow{\zeta} t, \emptyset, \kappa \cup \mathit{Eff}(t'' \leq t) \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}\rangle$
  (2) $\mu \models \kappa \cup \mathit{Eff}(t'' \leq t) \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$
  where $t' \xrightarrow{\zeta} t = \mathit{New}(\tau' \to \tau)$

  From (1), by the definition of $\mathcal{S}$
  (3) $\langle t'', c, \kappa\rangle = \mathcal{S}(\mathcal{E}[\mathtt{f} \mapsto t' \xrightarrow{\zeta} t][\mathtt{x} \mapsto t'])$

  From (2), by the definition of effect models
  (4) $\mu \models \kappa$
  (5) $\mu \models \mathit{Eff}(t'' \leq t)$
  (6) $\mu \models \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$, i.e. $\mu\zeta \supseteq \{\mathtt{n}\} \cup \mu c$

  From (3)(4), by induction
  (7) $\mu(\mathcal{E}[\mathtt{f} \mapsto t' \xrightarrow{\zeta} t][\mathtt{x} \mapsto t']) \vdash \mathtt{e} : \mu t'', \mu c$

  From (5), by Lemma 5.3
  (8) $\mu t'' \leq \mu t$

  From (7)(8), by (*sub*) in the static semantics
  (9) $\mu(\mathcal{E}[\mathtt{f} \mapsto t' \xrightarrow{\zeta} t][\mathtt{x} \mapsto t']) \vdash \mathtt{e} : \mu t, \mu c$

  From (9), by (*abs*) in the static semantics
  (10) $(\mu\mathcal{E})[\mathtt{f} \mapsto \mu(t' \xrightarrow{\zeta} t)] \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : \mu(t' \xrightarrow{\{\mathtt{n}\}\cup c} t), \emptyset$

  From (6), by the definition of subtype relation
  (11) $\mu(t' \xrightarrow{\{\mathtt{n}\}\cup c} t) \leq \mu(t' \xrightarrow{\zeta} t)$

From (10)(11), by (*sub*) rule in the static semantics
(12) $(\mu\mathcal{E})[\mathtt{f} \mapsto \mu(t' \xrightarrow{\zeta} t)] \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : \mu(t' \xrightarrow{\zeta} t), \emptyset$

From (12), by (*rec*) rule in the static semantics
$\mu\mathcal{E} \vdash (\mathtt{rec}\ \mathtt{n}\ (\mathtt{f}\ \mathtt{x})\ \mathtt{e}) : \mu(t' \xrightarrow{\zeta} t), \emptyset$

- Case of (*app*)

  The hypotheses are
  (1) $\mathcal{S}(\mathcal{E}, (\mathtt{e}\ \mathtt{e}')) = \langle t, c \cup c' \cup c'', \kappa \cup \kappa' \cup \mathit{Eff}(t' \leq t'')\rangle$
  (2) $\mu \models \kappa \cup \kappa' \cup \mathit{Eff}(t' \leq t'')$

  From (1), by the definition of $\mathcal{S}$
  (3) $\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t'' \xrightarrow{c''} t, c, \kappa\rangle$
  (4) $\mathcal{S}(\mathcal{E}, \mathtt{e}') = \langle t', c', \kappa'\rangle$

  From (2), by the definition of effect models
  (5) $\mu \models \kappa$
  (6) $\mu \models \kappa'$
  (7) $\mu \models \mathit{Eff}(t' \leq t'')$

  From (3)(5) and (4)(6), by induction
  (8) $\mu\mathcal{E} \vdash \mathtt{e} : \mu(t'' \xrightarrow{c''} t), \mu c$
  (9) $\mu\mathcal{E} \vdash \mathtt{e}' : \mu t', \mu c'$

  From (7), by Lemma 5.3
  (10) $\mu t' \leq \mu t''$

  From (9)(10), by the (*sub*) rule in the static semantics
  (11) $\mu\mathcal{E} \vdash \mathtt{e}' : \mu t'', \mu c'$

  From (8)(11), by (*app*) in the static semantics
  $\mu\mathcal{E} \vdash (\mathtt{e}\ \mathtt{e}') : \mu t, \mu(c \cup c' \cup c'')$ ♣

Proof of Theorem 5.2

Theorem 5.2 [Completeness] If $\theta_1\mathcal{E} \vdash \mathtt{e} : t_1, c_1$, then $\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t, c, \kappa\rangle$ and there exists a effect model μ of κ, such that:

$$\theta_1\mathcal{E} = \mu\mathcal{E} \quad \textit{and} \quad \mu t \leq t_1 \quad \textit{and} \quad c_1 \supseteq \mu c$$

Proof By induction on the structure of expressions

- Case of (var)

  The hypothesis is
  $\theta_1\mathcal{E} \vdash \mathtt{x} : t_1, \emptyset$

  By (var) and (sub) rules in the static semantics
  (1) $t_1' = \mathcal{E}(\mathtt{x})$
  (2) $\theta_1 t_1' \leq t_1$

  From (1), by the definition of $\mathcal{S}$
  $\mathcal{S}(\mathcal{E}, \mathtt{x}) = \langle t, \emptyset, \mathit{Eff}(t_1' \leq t)\rangle$
  where $t = \mathit{New}(\mathit{Erase}(t_1'))$

  Since t includes only fresh effect variables, taking θ, such that:
  (3) $\theta t = t_1$

  We define the effect model μ, such that :

$$\mu\vartheta = \begin{cases} \theta\vartheta & \vartheta \in \mathit{fv}(t) \\ \theta_1\vartheta & \mathit{otherwise} \end{cases}$$

  Note that since t includes only fresh effect variables, μ is well defined.

  From (2)(3), by the definition of μ
  (4) $\mu t_1' = \theta_1 t_1' \leq t_1$
  (5) $\mu t = \theta t = t_1$

  From (4)(5), by Lemma 5.3
  $\mu \models \mathit{Eff}(t_1' \leq t)$

  By the definition of μ
  $\theta_1\mathcal{E} = \mu\mathcal{E}$

  From (5)
  $\mu t \leq t_1$

- Case of (abs)

  The hypothesis is
  $\theta_1\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t_2' \stackrel{c_2}{\rightarrow} t_2, \emptyset$

  By (abs) and (sub) rule in the static semantics
  (1) $\theta_1\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t_1' \stackrel{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1, \emptyset$
  (2) $t_1' \stackrel{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1 \leq t_2' \stackrel{c_2}{\rightarrow} t_2$

From (1), by (abs) rule in the static semantics
(3) $(\theta_1\mathcal{E})[\mathtt{x} \mapsto t_1'] \vdash \mathtt{e} : t_1, c_1$

If $\mathtt{x}$ is of the principal type τ, let $t' = New(\tau)$
then there exists a substitution θ, such that:
(4) $t_1' = \theta t'$

we define a substitution θ_1', such that :

$$\theta_1'\vartheta = \begin{cases} \theta\vartheta & \vartheta \in fv(t') \\ \theta_1\vartheta & otherwise \end{cases}$$

Note that since t' includes only fresh effect variables, θ_1' is well defined.

From (4), by the definition of θ_1', (3) is equivalent with :
(5) $\theta_1'(\mathcal{E}[\mathtt{x} \mapsto t']) \vdash \mathtt{e} : t_1, c_1$

From (5), by induction
(6) $\mathcal{S}(\mathcal{E}[\mathtt{x} \mapsto t'], \mathtt{e}) = \langle t, c, \kappa \rangle$
$\exists\mu$, such that :
(7) $\mu \models \kappa$
(8) $\theta_1'(\mathcal{E}[\mathtt{x} \mapsto t']) = \mu(\mathcal{E}[\mathtt{x} \mapsto t'])$
(9) $\mu t \leq t_1$
(10) $c_1 \supseteq \mu c$

From (8)(4), by the definition of θ_1'
(11) $\theta_1\mathcal{E} = \mu\mathcal{E}$
(12) $t_1' = \mu t'$

From (6), since $t' = New(\tau)$, by the definition of $\mathcal{S}$
(13) $\mathcal{S}(\mathcal{E}, \lambda_{\mathtt{n}}\ (\mathtt{x} : \tau)\ \mathtt{e}) = \langle t' \xrightarrow{\zeta} t, \emptyset, \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\} \rangle$
where ζ *new*

We define a effect substitution μ' on $fv(\mathcal{E}, t', t, c, \kappa)$ and ζ, such that :

$$\mu'\vartheta = \begin{cases} \mu\vartheta & fv(\mathcal{E}, t', t, c, \kappa) \\ c_2 & \vartheta = \zeta \end{cases}$$

Note that since ζ is fresh, μ' is well defined.

From (7), by the definition of μ'
(14) $\mu' \models \kappa$

By the definition of μ'
(15) $\mu'\zeta = c_2$
(16) $\mu'(\{\mathtt{n}\} \cup c) = \{\mathtt{n}\} \cup \mu c$

From (10)(16), by the definition of μ'
(17) $\{\mathtt{n}\} \cup c_1 \supseteq \mu'(\{\mathtt{n}\} \cup c)$

From (2), by the definition of subtype relation
(18) $t_1 \leq t_2$
(19) $t_2' \leq t_1'$
(20) $c_2 \supseteq \{\mathtt{n}\} \cup c_1$

From (20)(15)(17), by the definition of effect models
(21) $\mu' \models \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$

From (14)(21), by the definition of effect models
$\mu' \models \kappa \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$

From (11)(12), by the definition of μ'
$\theta_1\mathcal{E} = \mu'\mathcal{E}$
(22) $t_1' = \mu' t'$

From (9), by the definition of μ'
(23) $\mu' t \leq t_1$

From (22)(23)(15), by the definition of subtype relation
(24) $\mu'(t' \xrightarrow{\zeta} t) \leq t_1' \xrightarrow{c_2} t_1$

From (18)(19), by the definition of subtype relation
(25) $t_1' \xrightarrow{c_2} t_1 \leq t_2' \xrightarrow{c_2} t_2$

From (24)(25)
$\mu'(t' \xrightarrow{\zeta} t) \leq t_2' \xrightarrow{c_2} t_2$

- Case of (*rec*)

  The hypothesis is
  $\theta_1\mathcal{E} \vdash (\mathtt{rec_n\ (f\ x)\ e}) : t_2' \xrightarrow{c_2} t_2, \emptyset$

  By (*rec*) and (*sub*) rule in the static semantics
  (1) $\theta_1\mathcal{E} \vdash (\mathtt{rec_n\ (f\ x)\ e}) : t_1' \overset{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1, \emptyset$
  (2) $t_1' \overset{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1 \leq t_2' \xrightarrow{c_2} t_2$

From (1), by (*rec*) rule in the static semantics
(3) $(\theta_1\mathcal{E})[\mathtt{f} \mapsto t_1' \stackrel{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1][\mathtt{x} \mapsto t_1'] \vdash \mathtt{e} : t_1, c_1$

If f and x is of the principal type $\tau' \rightarrow \tau$, τ' respectively,
let $(t' \stackrel{\zeta}{\rightarrow} t) = New(\tau' \rightarrow \tau)$, then there exists a substitution θ, such that:
(4) $t_1' \stackrel{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1 = \theta(t' \stackrel{\zeta}{\rightarrow} t)$

We define a substitution of effect variables θ_1', such that :

$$\theta_1'\vartheta = \begin{cases} \theta\vartheta & \vartheta \in fv(t' \stackrel{\zeta}{\rightarrow} t) \\ \theta_1\vartheta & otherwise \end{cases}$$

Note that since $t' \stackrel{\zeta}{\rightarrow} t$ includes only fresh effect variables, θ_1' is well defined.

From (4), by the definition of θ_1', (3) is equivalent with :
(5) $\theta_1'(\mathcal{E}[\mathtt{f} \mapsto t' \stackrel{\zeta}{\rightarrow} t][\mathtt{x} \mapsto t']) \vdash \mathtt{e} : t_1, c_1$

From (5), by induction
(6) $\mathcal{S}(\mathcal{E}[\mathtt{f} \mapsto t' \stackrel{\zeta}{\rightarrow} t][\mathtt{x} \mapsto t'], \mathtt{e}) = \langle t'', c, \kappa \rangle$
$\exists\mu$, such that :
(7) $\mu \models \kappa$
(8) $\theta_1'(\mathcal{E}[\mathtt{f} \mapsto t' \stackrel{\zeta}{\rightarrow} t][\mathtt{x} \mapsto t']) = \mu(\mathcal{E}[\mathtt{f} \mapsto t' \stackrel{\zeta}{\rightarrow} t][\mathtt{x} \mapsto t'])$
(9) $\mu t'' \leq t_1$
(10) $c_1 \supseteq \mu c$

From (6), since $t' \stackrel{\zeta}{\rightarrow} t = New(\tau' \rightarrow \tau)$, by the definition of $\mathcal{S}$
(11) $\mathcal{S}(\mathcal{E}, (\mathtt{rec_n}\ (\mathtt{f} : \tau' \rightarrow \tau\ \mathtt{x} : \tau')\ \mathtt{e})) = \langle t' \stackrel{\zeta}{\rightarrow} t, \emptyset, \kappa \cup Eff(t'' \leq t) \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\} \rangle$

From (8)(4), by the definition of θ_1'
$\theta_1\mathcal{E} = \mu\mathcal{E}$
(12) $t_1' \stackrel{\{\mathtt{n}\}\cup c_1}{\rightarrow} t_1 = \mu(t' \stackrel{\zeta}{\rightarrow} t)$
(13) $t_1 = \mu t$
(14) $\{\mathtt{n}\} \cup c_1 = \mu\zeta$

From (9)(13), by Lemma 5.3
(15) $\mu \models Eff(t'' \leq t)$

From (14)(10), by the definition of effect models
(16) $\mu \models \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$

From (7)(15)(16), by the definition of effect models
$\mu \models \kappa \cup \mathit{Eff}(t'' \leq t) \cup \{\zeta \supseteq \{\mathtt{n}\} \cup c\}$

From (12)(2)
$\mu(t' \xrightarrow{\zeta} t) \leq t'_2 \xrightarrow{c_2} t_2$

- Case of (app)

  The hypotheses is
  $\theta_1\mathcal{E} \vdash (\mathtt{e}\ \mathtt{e}') : t_2, c_1 \cup c'_1 \cup c''_1$

  By (app) and (sub) rules in the static semantics
  (1) $\theta_1\mathcal{E} \vdash (\mathtt{e}\ \mathtt{e}') : t_1, c_1 \cup c'_1 \cup c''_1$
  (2) $t_1 \leq t_2$

  From (1), by (app) in the static semantics
  (3) $\theta_1\mathcal{E} \vdash \mathtt{e} : t'_1 \xrightarrow{c''_1} t_1, c_1$
  (4) $\theta_1\mathcal{E} \vdash \mathtt{e}' : t'_1, c'_1$

  From (3), by induction
  (5) $\mathcal{S}(\mathcal{E}, \mathtt{e}) = \langle t'' \xrightarrow{c''} t, c, \kappa \rangle$
  $\exists \mu$, such that :
  (6) $\mu \models \kappa$
  (7) $\theta_1\mathcal{E} = \mu\mathcal{E}$
  (8) $\mu(t'' \xrightarrow{c''} t) \leq t'_1 \xrightarrow{c''_1} t_1$
  (9) $c_1 \supseteq \mu c$

  From (4), by induction
  (10) $\mathcal{S}(\mathcal{E}, \mathtt{e}') = \langle t', c', \kappa \rangle$
  $\exists \mu'$, such that :
  (11) $\mu' \models \kappa'$
  (12) $\theta_1\mathcal{E} = \mu'\mathcal{E}$
  (13) $\mu' t' \leq t'_1$
  (14) $c'_1 \supseteq \mu' c'$

  From (5)(10), by the definition of $\mathcal{S}$
  $\mathcal{S}(\mathcal{E}, (\mathtt{e}\ \mathtt{e}')) = \langle t, c \cup c' \cup c'', \kappa \cup \kappa' \cup \mathit{Eff}(t' \leq t'') \rangle$

  We define a substitution μ'' on $\mathit{fv}(\mathcal{E}, t'' \xrightarrow{c''} t, c, \kappa)$, and $\mathit{fv}(\mathcal{E}, t', c', \kappa')$

$$\mu''\vartheta = \begin{cases} \mu\vartheta & \vartheta \in \mathit{fv}(\mathcal{E}, t'' \xrightarrow{c''} t, c, \kappa) \\ \mu'\vartheta & \vartheta \in \mathit{fv}(\mathcal{E}, t', c', \kappa') \end{cases}$$

Note that $\forall\vartheta$, if $\vartheta \in fv(\mathcal{E}, t', c', \kappa')$ and $\vartheta \in fv(\mathcal{E}, t, c, \kappa)$
then by the definition of $\mathcal{S}$, $\vartheta \in fv(\mathcal{E})$
By (7)(12), $\mu\vartheta = \mu'\vartheta$, thus μ'' is well defined.

From (6)(11), by the definition of μ''
(16) $\mu'' \models \kappa$
(17) $\mu'' \models \kappa'$

From (8), by the definition of μ''
(18) $\mu''(t'' \xrightarrow{c''} t) = \mu(t'' \xrightarrow{c''} t) \leq t_1' \xrightarrow{c_1''} t_1$

From (18), by the definition of subtype relation
(19) $\mu''t \leq t_1$
(20) $t_1' \leq \mu''t''$
(21) $c_1'' \supseteq \mu''c''$

From (13), by the definition of μ''
(22) $\mu''t' = \mu't' \leq t_1'$

From (20)(22), by Lemma 5.3
(23) $\mu'' \models \mathit{Eff}(t' \leq t'')$

From (16)(17)(23), by the definition of effect models
$\mu'' \models \kappa \cup \kappa' \cup \mathit{Eff}(t' \leq t'')$

From (19)(2)
$\mu''t \leq t_2$

From (9)(14)(21), by the definition of μ''
$c_1 \cup c_1' \cup c_1'' \supseteq \mu''(c \cup c' \cup c'')$ ♣

Appendix 4

Proof of Theorem 6.1

Theorem 6.1 [Well-Formedness of Abstract Semantics] If $\hat{\beta}, \hat{E} \vdash \mathtt{e} \rightarrow \hat{v}, \hat{c}$ and $\mathcal{WF}(\hat{\beta}, \hat{E})$, then

- $(\hat{v}, \hat{E})$ is well-formed.
- All $(\hat{\beta}', \hat{E}')$ used in the $\rightarrow$ derivation tree of $\mathtt{e}$ are well-formed.

Proof By induction on the number of reduction steps of expressions.

- The hypotheses are
  (1) $\hat{\beta}, \hat{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \rightarrow \cup_{i=1}^{r} \hat{v}_i, \cup_{i=1}^{r}(\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})$
  (2) $\mathcal{WF}(\hat{\beta}, \hat{E})$

  From hypothesis (1), by (app) in abstract semantics
  (3) $\hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\}$
  (4) $\hat{\beta}, \hat{E} \vdash \mathtt{a}' \rightarrow \hat{v}'$
  (5) $\hat{\beta}_i'[\mathtt{x}_i \mapsto \hat{b}'], \hat{E} \vdash \mathtt{e}_i' \rightarrow \hat{v}_i, \hat{c}_i$
  (6) $\hat{E}(\mathtt{x}_i, \hat{b}') = \hat{v}'$
  where $\hat{b}' = \mathtt{l}$ and $i = 1 \ldots r$

  From (2)(3)(4), by Lemma 6.1
  (7) $\mathcal{WF}(\{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\}, \hat{E})$
  (8) $\mathcal{WF}(\hat{v}', \hat{E})$

  From (7), by Definition 6.1
  (9) $\mathcal{WF}(\hat{\beta}_i', \hat{E})$

  From (6)(8)(9), by Definition 6.1
  (10) $\mathcal{WF}(\hat{\beta}_i'[\mathtt{x}_i \mapsto \hat{b}'], \hat{E})$

  From (10), by induction
  All $(\hat{\beta}', \hat{E}')$ used in the $\rightarrow$ derivation tree of $\mathtt{e}$ are well-formed

From (5)(10), by induction
(11) $(\hat{v}_i, \hat{E})$ is well-formed

From (11), by Definition 6.1
$(\cup_{i=1}^{r} \hat{v}_i, \hat{E})$ is well-formed ♣

Proof of Theorem 6.2

Theorem 6.2 [Consistency of Abstract Semantics]

$$\left.\begin{array}{l} b, \beta, E \vdash \mathtt{e} \rightarrow v, c \\ \hat{\beta}, \hat{E} \vdash \mathtt{e} \rightarrow \hat{v}, \hat{c} \\ (\beta, E) \leq (\hat{\beta}, \hat{E}) \end{array}\right\} \Rightarrow \left\{\begin{array}{l} (v, E) \leq (\hat{v}, \hat{E}) \\ c \sqsubseteq \hat{c} \end{array}\right.$$

Proof By induction on the number of reduction steps of expressions.

- The hypotheses are
  (1) $(\beta, E) \leq (\hat{\beta}, \hat{E})$
  (2) $b, \beta, E \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \rightarrow v, c \cup \{(\mathtt{l}, \beta) \rightsquigarrow \{\mathtt{n}\}\}$
  (3) $\hat{\beta}, \hat{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \rightarrow \cup_{i=1}^{r} \hat{v}_i, \cup_{i=1}^{r} (\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})$

  From hypothesis (2), by (*app*) in the dynamic semantics
  (4) $\beta, E \vdash \mathtt{a} \rightarrow (\lambda_{\mathtt{n}}(\mathtt{x})\ \mathtt{e}, \beta')$
  (5) $\beta, E \vdash \mathtt{a}' \rightarrow v'$
  (6) $b', \beta'[\mathtt{x} \mapsto b'], E \vdash \mathtt{e} \rightarrow v, c$
  (7) $E(\mathtt{x}, b') = v'$
  where $b' = b.\mathtt{l}$

  From hypothesis (3), by (*app*) in the abstract semantics
  (8) $\hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i')\}$
  (9) $\hat{\beta}, \hat{E} \vdash \mathtt{a}' \rightarrow \hat{v}'$
  (10) $\hat{\beta}_i'[\mathtt{x}_i \mapsto \hat{b}'], \hat{E} \vdash \mathtt{e}_i \rightarrow \hat{v}_i, \hat{c}_i$
  (11) $\hat{E}(\mathtt{x}_i, \hat{b}') = \hat{v}'$
  where $\hat{b}' = \mathtt{l}$ and $i = 1 \ldots r$

  From (1)(4)(8) and (1)(5)(9), by Lemma 6.2
  (12) $((\lambda_{\mathtt{n}}(\mathtt{x})\ \mathtt{e}, \beta'), E) \leq (\{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\}, \hat{E})$
  (13) $(v', E) \leq (\hat{v}', \hat{E})$

  From (12), by Definition 6.2, $\exists j\ (1 \leq j \leq r)$ *s.t.*
  (14) $\lambda_{\mathtt{n}}(\mathtt{x})\ \mathtt{e} = \lambda_{\mathtt{n}_j}(\mathtt{x}_j)\ \mathtt{e}_j$
  (15) $(\beta', E) \leq (\hat{\beta}_j', \hat{E})$

From (15)(7)(11)(13), by Definition 6.2
(16) $(\beta'[\mathtt{x} \mapsto b'], E) \leq (\hat{\beta}_j'[\mathtt{x}_j \mapsto \hat{b}'], \hat{E})$

From (16)(6)(10), by induction
(17) $(v, E) \leq (\hat{v}_j, \hat{E})$
(17)′ $c \sqsubseteq \hat{c}_j$

From (17), by Definition 6.2
$(v, E) \leq (\cup_{i=1}^{r} \hat{v}_i, \hat{E})$

From (14)(17)′, by Definition 6.3
$c \cup \{(\mathtt{l}, \beta) \rightsquigarrow \{\mathtt{n}\}\} \sqsubseteq \cup_{i=1}^{r}(\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})$ ♣

Proof of Theorem 6.3

Theorem 6.3 [Types of Abstract Semantics]

$$\left.\begin{array}{l} \mathcal{E} \vdash \mathtt{e} : t, \bar{c} \\ \hat{\beta}, \hat{E} \vdash \mathtt{e} \rightarrow \hat{v}, \hat{c} \\ (\hat{\beta}, \hat{E}) : \mathcal{E} \end{array}\right\} \Rightarrow \left\{\begin{array}{l} (\hat{v}, \hat{E}) : t \\ \hat{c} \sqsubseteq \bar{c} \end{array}\right.$$

Proof By induction on the number of reduction steps of expressions.

- The hypotheses are
  (1) $(\hat{\beta}, \hat{E}) : \mathcal{E}$
  (2) $\mathcal{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} : t, \cup_{(\mathtt{n}, \bar{c}) \in d}(\bar{c} \cup \{(\mathtt{l}, []) \rightsquigarrow \{\mathtt{n}\}\})$
  (3) $\hat{\beta}, \hat{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \rightarrow \cup_{i=1}^{r} \hat{v}_i, \cup_{i=1}^{r}(\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})$

  From (2), by (*app*) in the type semantics
  (4) $\mathcal{E} \vdash \mathtt{a} : t' \xrightarrow{d} t$
  (5) $\mathcal{E} \vdash \mathtt{a}' : t'$

  From (3), by (*app*) in the abstract semantics
  (6) $\hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\}$
  (7) $\hat{\beta}, \hat{E} \vdash \mathtt{a}' \rightarrow \hat{v}'$
  (8) $\hat{\beta}_i'[\mathtt{x}_i \mapsto \hat{b}'], \hat{E} \vdash \mathtt{e}_i \rightarrow \hat{v}_i, \hat{c}_i$
  (9) $\hat{E}(\mathtt{x}_i, \hat{b}') = \hat{v}'$
  where $\hat{b}' = \mathtt{l}$ and $i = 1 \ldots r$

  From (1)(4)(6) and (1)(5)(7), by Lemma 6.3
  (10) $(\{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\}, \hat{E}) : t' \xrightarrow{d} t$
  (11) $(\hat{v}', \hat{E}) : t'$

From (10), by Definition 6.4, $\forall i\ (i = 1 \ldots r)$, $\exists \mathcal{E}'_i$ *s.t.*
(12) $(\hat{\beta}'_i, \hat{E}) : \mathcal{E}'_i$
(13) $\mathcal{E}'_i \vdash (\lambda_{\mathtt{n}_i}\ (\mathtt{x}_i)\ \mathtt{e}_i) : t' \xrightarrow{d} t$

From (13), by (*abs*) in the type semantics
(14) $\mathcal{E}'_{i\mathtt{x}_i}[\mathtt{x}_i \mapsto t'] \vdash \mathtt{e}_i : t, \bar{c}_i$
(15) $(\mathtt{n}_i, \bar{c}_i) \in d$

From (12)(11)(9), by Definition 6.4
(16) $(\hat{\beta}'_i[\mathtt{x}_i \mapsto \hat{b}'], \hat{E}) : \mathcal{E}'_{i\mathtt{x}_i}[\mathtt{x}_i \mapsto t']$

From (16)(8)(14), by induction
(17) $(\hat{v}_i, \hat{E}) : t$
(17)′ $\hat{c}_i \sqsubseteq \bar{c}_i$

From (17), by Definition 6.4
$(\cup_{i=1}^{r} \hat{v}_i, \hat{E}) : t$

From (15)(17)′, by Definition 6.3
$\cup_{i=1}^{r}(\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\}) \sqsubseteq \cup_{(\mathtt{n},\bar{c})\in d}(\bar{c} \cup \{(\mathtt{l}, []) \rightsquigarrow \{\mathtt{n}\}\})$ ♣

Proof of Lemma 6.4

Lemma 6.4 [Well-Formedness of $\mathcal{A}(t)$] $\mathcal{A}(t)$ is well-formed.
Proof By induction on the structure of types

- Case int

  By the definition of $\mathcal{A}$
  $\mathcal{A}(int) = (\emptyset, [])$

  By Definition 6.1
  $\mathcal{A}(int)$ is well-formed

- Case $(t' * t_0) \xrightarrow{d} t_1$ where $t_0 = t \xrightarrow{d'} t_1$

  By the definition of $\mathcal{A}$
  $\mathcal{A}((t' * t_0) \xrightarrow{d} t_1) = (\hat{v}', \hat{E}')$
  where
  (1) $\hat{v}' = \{(\lambda_{\mathtt{n}_i}(\mathtt{x}\ \mathtt{k}_i)\ \mathtt{e}_i, [\mathtt{x}_i \mapsto \mathtt{l}_i]) \mid i = 1 \ldots q\}$
  (2) $\hat{E}' = \hat{E}[(\mathtt{x}_i, \mathtt{l}_i) \mapsto \hat{v}]$
  (3) $(\hat{v}, \hat{E}) = \mathcal{A}(t)$

From (3), by induction
(4) $\mathcal{A}(t)$ is well-formed, i.e. $(\hat{v},\hat{E})$ is well-formed

From (2)(4), since $\mathtt{x}_i$ is fresh , by Definition 6.1
(5) $(\hat{v},\hat{E}')$ is well-formed

From (5), by Definition 6.1
(6) $([\mathtt{x}_i \mapsto \mathtt{l}_i],\hat{E}')$ is well-formed

From (6)(1)(2), by Definition 6.1
$\mathcal{A}((t' * t_0) \xrightarrow{d} t_1)$ is well-formed ♣

Proof of Lemma 6.5

Lemma 6.5 [Consistency of $\mathcal{A}(t)$] If $(\hat{v},\hat{E}) : t$, then $(\hat{v},\hat{E}) \leq \mathcal{A}(t)$
Proof By induction on the structure of types

- Case int

  By the definition of $\mathcal{A}$
  (1) $\mathcal{A}(int) = (\emptyset, [])$

  Since $(\hat{v},\hat{E}) : t$, by Definition 6.4
  (2) $(\emptyset,\hat{E}) : int$

  From (1)(2), by Definition 6.5
  $(\emptyset,\hat{E}) \leq \mathcal{A}(int)$

- Case $(t' * t_0) \xrightarrow{d} t_1$

  By the definition of $\mathcal{A}$

  (1) $\mathcal{A}((t' * t_0) \xrightarrow{d} t_1) = (\hat{v}',\hat{E}')$
  where
  $\hat{v}' = \{(\lambda_{\mathtt{n}_i}(\mathtt{x}\ \mathtt{k}_i)\ \mathtt{e}_i, [\mathtt{x}_i \mapsto \mathtt{l}_i]) \mid i = 1 \ldots q\}$
  $d = \{(\mathtt{n}_i, \bar{c}_i) \mid i = 1 \ldots q\}$

  Since $(\hat{v},\hat{E}) : t$, by Definition 6.4
  (2) $\forall(\mathtt{n},\hat{\beta}) \in \hat{v},\ \exists\mathcal{E},\ s.t.\ (\hat{\beta},\hat{E}) : \mathcal{E}$
  (3) $\mathcal{E} \vdash \mathtt{n} : t$

  From (3), by the (abs) rule in the effect semantics
  (4) $\exists\bar{c},\ s.t.\ (\mathtt{n},\bar{c}) \in d$

From (1)(2)(4), by Definition 6.5
$(\hat{v}, \hat{E}) \leq \mathcal{A}(t)$

Proof of Lemma 6.6

Lemma 6.6 [Simulation] For any $\hat{\beta}_1$ and $\hat{E}_1$, if

$$\hat{\beta}_1[\mathtt{x}_i \mapsto \mathtt{l}_i][\mathtt{k}_i \mapsto \mathtt{l}_k], \hat{E}_1[(\mathtt{x}_i, \mathtt{l}_i) \mapsto \hat{v}][(\mathtt{k}_i, \mathtt{l}_k) \mapsto \{Id\}] \vdash \mathcal{S}(\bar{c}_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \rightarrow \hat{v}, \hat{c}_i$$

then $\mathcal{D}(\hat{c}_i) = \bar{c}_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\}$.
Proof By induction on $\bar{c}_i$

Supposing $\forall j \geq 1$
$\hat{\beta}'_j = \hat{\beta}_j[\mathtt{x}_i \mapsto \mathtt{l}_i][\mathtt{k}_i \mapsto \mathtt{l}_k]$
$\hat{E}'_j = \hat{E}_j[(\mathtt{x}_i, \mathtt{l}_i) \mapsto \hat{v}][(\mathtt{k}_i, \mathtt{l}_k) \mapsto \{Id\}]$

- Case $\bar{c}_i = []$

  By the definition of $\mathcal{S}$
  $\mathcal{S}([], \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) = (\mathtt{k}_i\ \mathtt{x}_i)_{\mathtt{l}}$

  By the abstract semantics, since $\mathtt{k}_i$ is bound to Id
  $\hat{\beta}'_1, \hat{E}'_1 \vdash (\mathtt{k}_i\ \mathtt{x}_i)_{\mathtt{l}} \rightarrow \hat{v}, \{(\mathtt{l}, \hat{\beta}'_1) \rightsquigarrow \{Id\}\}$

  By the definition of $\mathcal{D}$
  $\mathcal{D}(\{(\mathtt{l}, \hat{\beta}'_1) \rightsquigarrow \{Id\}\}) = \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\}$

- Case $\bar{c}_i = \bar{c}'_i \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_1 \ldots \mathtt{n}_r\}\}$

  By the definition of $\mathcal{S}$
  (1) $\mathcal{S}(\bar{c}_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) = \mathcal{S}'(\{\mathtt{n}_1 \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l})$

  By the definition of $\mathcal{S}'$, $\forall j = 1 \ldots r$
  (2) $\mathcal{S}'(\{\mathtt{n}_j \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) = (\ (\lambda_{\mathtt{n}_j}(\mathtt{k}_j)\, \mathcal{S}'(\{\mathtt{n}_{j+1} \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_k, \mathtt{x}_i, \mathtt{l}))\, \mathtt{k}_i\ )_{\mathtt{l}'}$
  Note that $\{\mathtt{n}_{j+1} \ldots \mathtt{n}_r\} = \emptyset$

  By induction on j, we prove that
  $\hat{\beta}'_j, \hat{E}'_j \vdash \mathcal{S}'(\{\mathtt{n}_j \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \rightarrow \hat{v}, \hat{c}'$ such that
  $\mathcal{D}(\hat{c}') = \bar{c}'_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\} \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_j \ldots \mathtt{n}_r\}\}$
  where
  $\hat{c}' = \hat{c}'_i \cup \{(\mathtt{l}', \hat{\beta}'_j) \rightsquigarrow \{\mathtt{n}_j\}\} \ldots \{(\mathtt{l}', \hat{\beta}'_r) \rightsquigarrow \{\mathtt{n}_r\}\}$
  $\hat{\beta}_{j+1} = \hat{\beta}_j[\mathtt{k}_j \mapsto \mathtt{l}']$
  $\hat{E}_{j+1} = \hat{E}_j[(\mathtt{k}_j, \mathtt{l}') \mapsto \mathtt{k}_i]$

– Case $j = r + 1$
By the definition of $\mathcal{S}'$, since $\{\mathtt{n}_{r+1} \ldots \mathtt{n}_r\} = \emptyset$
(3) $\mathcal{S}'(\emptyset, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) = \mathcal{S}(\bar{c}'_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l})$

From (3), by the induction on $\bar{c}_i$
$\hat{\beta}'_{r+1}, \hat{E}'_{r+1} \vdash \mathcal{S}'(\emptyset, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \rightarrow \hat{v}, \hat{c}'_i$ such that
$\mathcal{D}(\hat{c}'_i) = \bar{c}'_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\}$

– Case $j = 1 \ldots r$
By induction on j
(4) $\hat{\beta}'_{j+1}, \hat{E}'_{j+1} \vdash \mathcal{S}'(\{\mathtt{n}_{j+1} \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \rightarrow \hat{v}, \hat{c}'$ such that
(5) $\mathcal{D}(\hat{c}') = \bar{c}'_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\} \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_{j+1} \ldots \mathtt{n}_r\}\}$
where
$\hat{c}' = \hat{c}'_i \cup \{(\mathtt{l}', \hat{\beta}'_{j+1}) \rightsquigarrow \{\mathtt{n}_{j+1}\}\} \ldots \{(\mathtt{l}', \hat{\beta}'_r) \rightsquigarrow \{\mathtt{n}_r\}\}$

From (2)(4), by the abstract semantics
$\hat{\beta}'_j, \hat{E}'_j \vdash \mathcal{S}'(\{\mathtt{n}_j \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \rightarrow \hat{v}, \hat{c}' \cup \{(\mathtt{l}', \hat{\beta}'_j) \rightsquigarrow \{\mathtt{n}_j\}\}$

From (5), by the definition of $\mathcal{D}$
$\mathcal{D}(\hat{c}' \cup \{(\mathtt{l}', \hat{\beta}'_j) \rightsquigarrow \{\mathtt{n}_j\}\}) = \bar{c}'_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\} \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_j \ldots \mathtt{n}_r\}\}$

From (1), using the initial abstract value environments $\hat{\beta}'_1$ and $\hat{E}'_1$, we get :
$\hat{\beta}'_1, \hat{E}'_1 \vdash \mathcal{S}(\bar{c}_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \rightarrow \hat{v}, \hat{c}_i$ such that
$\mathcal{D}(\hat{c}_i) = \bar{c}_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\}$
where
$\hat{c}_i = \hat{c}'_i \cup \{(\mathtt{l}', \hat{\beta}'_1) \rightsquigarrow \{\mathtt{n}_1\}\} \ldots \{(\mathtt{l}', \hat{\beta}'_r) \rightsquigarrow \{\mathtt{n}_r\}\}$ ♣

Proof of Lemma 6.7

Lemma 6.7 [Well-Formedness of $(\hat{\beta}_0, \hat{E}_0)$] $(\hat{\beta}_0, \hat{E}_0)$ is well-formed.
Proof

- By the definition of $(\hat{\beta}_0, \hat{E}_0)$
(1) $\forall \mathtt{x} \in Dom(\hat{\beta}_0),\ (\mathtt{x}, \hat{\beta}_0(\mathtt{x})) \in Dom(\hat{E}_0)$
(2) $\hat{E}_0(\mathtt{x}, \hat{\beta}_0(\mathtt{x})) = \hat{v}$
(3) $Dom(\hat{E}) \subseteq Dom(\hat{E}_0)$
(4) $(\hat{v}, \hat{E}) = \mathcal{A}(\mathcal{E}(\mathtt{x}))$

From (4), by Lemma 6.4
(5) $(\hat{v}, \hat{E})$ is well-formed

From (5)(3), by Definition 6.1
(6) $(\hat{v}, \hat{E}_0)$ is well-formed

From (1)(2)(6), by Definition 6.1
$(\hat{\beta}_0, \hat{E}_0)$ is well-formed ♣

Proof of Lemma 6.8

Lemma 6.8 [Consistency of $(\hat{\beta}_0, \hat{E}_0)$] If $(\hat{\beta}'_0, \hat{E}'_0) : \mathcal{E}$, then $(\hat{\beta}'_0, \hat{E}'_0) \leq (\hat{\beta}_0, \hat{E}_0)$
Proof

- By the definition of $(\hat{\beta}_0, \hat{E}_0)$
  (1) $\forall \mathtt{x} \in Dom(\mathcal{E}),\ \mathtt{x} \in Dom(\hat{\beta}_0)$
  (2) $\hat{E}_0(\mathtt{x}, \hat{\beta}_0(\mathtt{x})) = \hat{v}$
  (3) $Dom(\hat{E}) \subseteq Dom(\hat{E}_0)$
  (4) $(\hat{v}, \hat{E}) = \mathcal{A}(\mathcal{E}(\mathtt{x}))$

  Since $(\hat{\beta}'_0, \hat{E}'_0) : \mathcal{E}$, by Definition 6.4
  (5) $\forall \mathtt{x} \in Dom(\hat{\beta}'_0),\ \mathtt{x} \in Dom(\mathcal{E})$
  (6) $(\hat{E}'_0(\mathtt{x}, \hat{\beta}'_0(\mathtt{x})), \hat{E}'_0) : \mathcal{E}(\mathtt{x})$

  From (6), by Lemma 6.5
  (7) $(\hat{E}'_0(\mathtt{x}, \hat{\beta}'_0(\mathtt{x})), \hat{E}'_0) \leq \mathcal{A}(\mathcal{E}(\mathtt{x}))$

  From (7)(4)
  (8) $(\hat{E}'_0(\mathtt{x}, \hat{\beta}'_0(\mathtt{x})), \hat{E}'_0) \leq (\hat{v}, \hat{E})$

  From (8)(3), by Definition 6.2
  (9) $(\hat{E}'_0(\mathtt{x}, \hat{\beta}'_0(\mathtt{x})), \hat{E}'_0) \leq (\hat{v}, \hat{E}_0)$

  From (5)(1)
  (10) $\forall \mathtt{x} \in Dom(\hat{\beta}'_0),\ \mathtt{x} \in Dom(\hat{\beta}_0)$

  From (10)(9)(2), by Definition 6.2
  $(\hat{\beta}'_0, \hat{E}'_0) \leq (\hat{\beta}_0, \hat{E}_0)$ ♣