A type system for embedded rewriting programming

Claudia Fernanda Oliveira Kiermes Tavares

Institut National de Recherche en Informatique et Automatique — Université de Lorraine
École doctorale IAEM Informatique, Automatique, Électronique et Mathématiques

Un système de types pour la programmation par réécriture embarquée

THÈSE

présentée et soutenue publiquement le 2 mars 2012

pour l'obtention du

Doctorat de l'Université de Lorraine

(spécialité informatique)

par

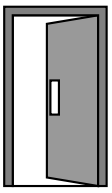
Cláudia Fernanda Oliveira Kiermes Tavares

Composition du jury

Rapporteurs :	Luigi Liquori	Directeur de Recherche, Inria, France
	Anamaria Martins Moreira	Professeur, UFRN, Brésil
Examinateurs :	Adam Cichon	Professeur, Université de Lorraine, France
	Rachid Echahed	Chargé de Recherche (Hdr), CNRS, France
Directeurs de thèse :	Claude Kirchner	Directeur de Recherche, Inria, France
	Pierre-Etienne Moreau	Professeur, École des Mines de Nancy, France

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

À mamãe, Ana Lígia, a papai, Orcines, e aos meus irmãos, Cláudia e Alexandre.
À memória do meu tio e padrinho Fernando (1959-2011) e da minha sobrinha Anna Karenina (2003-2011) que nos deixaram de maneira tão inesperada.



Remerciements

Il y a quatre ans et demi, je suis partie du Brésil avec un rêve de connaître le monde. Je rêvais aussi de faire une thèse dans un laboratoire de renommée internationale dans le domaine des méthodes formelles. Je me suis lancée dans ce défi avec le soutien de mes directeurs de thèse, Claude Kirchner et Pierre-Etienne Moreau, que je remercie ici pour la rigueur scientifique qu'ils m'ont apportée ainsi que pour tous leurs conseils pertinents. Ils ont su suivre ma progression avec beaucoup de patience. Leur encadrement m'a beaucoup appris, notamment à être plus objective et à me concentrer sur les points les plus importants de ce travail malgré mes difficultés d'adaptation. Ceci me servira bien au-delà de mon doctorat.

Je voudrais remercier les personnes qui m'ont fait l'honneur d'accepter d'être membre de mon jury de soutenance. Anamaria Moreira et Luigi Liquori ont pris le temps d'être rapporteurs de ce manuscrit et de lui apporter des contributions minutieuses et importantes. Merci en particulier à Anamaria qui accompagne mon parcours universitaire depuis mes études à la faculté d'informatique au Brésil. Je remercie Rachid Echahed pour ses questions pertinentes qui m'ont poussée à réfléchir à propos de l'application de mon travail aux termes-graphes. Je remercie aussi Adam Cichon d'avoir accepté de présider mon jury.

Durant mes années de thèse j'ai eu l'occasion de changer de bureau quelques fois tout en profitant de la bonne ambiance qui régnait dans l'équipe Pareo (nommée encore Protheo lorsque j'ai commencé). Je tiens à remercier Horatiu Cirstea, Sorin Stratulat et tous mes collègues et amis d'équipe. Merci à mon concitoyen Anderson, qui m'a introduit auprès de cette équipe et m'a accueilli chez lui pendant mes premiers jours en France. Je remercie mes premiers collègues de bureau Clément, Paul et Cody de m'avoir offert un environnement de travail joyeux et amusant. Ils ont facilité mon adaptation à la culture française avec beaucoup d'humour et d'amitié. Les bandes dessinées de Clément étaient toujours drôles. L'encouragement de Cody ainsi que nos échanges me donnaient le moral pour persister dans ce défi. C'est par ailleurs Paul qui a le plus contribué à mon initiation aux systèmes de types et à la programmation en Tom.

Merci à Tony et Oana avec qui j'ai également partagé un bureau. Par son amitié et son sens de l'écoute, Oana a toujours compris mes angoisses et a su m'apporter le soutien dont j'avais besoin. Merci à François et Jean-Christophe, mes derniers collègues de bureau. En plus de m'avoir transmis certaines des ses compétences en développement logiciel, Jean-Christophe a partagé avec moi des bons et des mauvais moments au bureau et même ailleurs. Avec une patience saisissante et un gros pot de bonbons, il a été essentiel à la fin de ce défi et son amitié m'est précieuse. Je remercie encore Émilie de m'avoir toujours motivé et donné ses points de vues originaux sur mon travail. Merci encore à ceux que j'ai côtoyés en début de thèse pour leur gentillesse que je conserve toujours vivante : Florent, Yves, Guillaume, Radu et Laura. Je leur souhaite à tous réussite et bonheur.

Pour terminer ce défi humain néanmoins professionnel, je tiens à remercier la

CAPES (l'Agence Brésilienne de Financement de la Recherche sous dossier BEX 4878-06-0) de m'avoir donné toutes les conditions nécessaires pour le développement de ce travail. J'adresse également mes remerciements à David Déharbe qui a accepté d'être mon tuteur vis-à-vis de la CAPES et qui m'a communiqué le goût pour les méthodes formelles.

Ces années de travail à plus de sept mille kilomètres loin de mon pays n'auraient pu être effectuées sans le soutien extérieur et sans borne de ma famille et de mes amis. Merci d'abord à mes parents Ana Lígia et Orcines, à ma sœur Anna Cláudia et à mon frère Alexandre d'être toujours là pour moi. Je les remercie pour leur compréhension et sacrifices lors des évènements douloureux de nos vies. Ma famille a fait preuve de beaucoup de courage lors de la récente perte de mon oncle et parrain Fernando et du violent accident qui a côuté la vie de ma petite cousine Anna Karenina et qui a mené mon petit cousin Otávio au coma. Le soutien infini de ma famille a été crucial pour mon bien-être émotionnel et l'achèvement de ce travail. Je remercie aussi la gentillesse et la disponibilité de mon oncle Pedro et de ma tante Petra qui m'ont accueilli chaleureusement à certaines reprises à Munich et m'ont souvent gâté.

Mon séjour en France aurait été bien plus difficile sans la complicité de Richard. Je le remercie de tout mon cœur pour sa patience et son soutien surtout durant la rédaction de ce manuscrit. Il a partagé au quotidien avec moi, tous mes doutes, mes peurs et mes joies avec des petits gâteaux d'encouragement et un interêt émouvant. Il a su par son amour égayer chaque jour passé pour préparer cette thèse. Sa famille aussi m'a accueillie à bras ouverts et j'en suis très reconnaissante. Merci pour les découvertes inoubliables avec Mireille, Alain, Leïla et Aldo. Cette famille m'a bien gâtée et m'a offert des très bons moments. Merci pour la tendresse habituelle de Nanou et Walter qui m'a été d'une grande valeur.

Merci à Carol, ma colocataire virtuelle, psychologue personnelle et gratuite dont les discussions m'apportaient un grand réconfort. Merci aussi à Diego pour tous les bons moments passés à Nancy. Merci pour les moments de détentes avec les filles de l'équipe R1 de volley-ball de Vandœuvre pendant trois saisons ainsi que pour leur amitié. Merci enfin à tous ceux que j'ai pu oublier et que je ne saurais citer ici. Ce doctorat a été le plus grand et passionnant défi que j'ai entrepris durant toute ma vie.

Contents

List of Figures

Résumé Étendu

Introduction

Dans le domaine du génie logiciel, les méthodes formelles ont été développées pour aider à garantir que le comportement d'un système correspond exactement aux attentes du programmeur. Le comportement souhaité doit être fourni à travers une spécification à prouver par la suite afin d'assurer que le système se comporte comme prévu. Cependant, les spécifications doivent être (au moins partiellement) écrites manuellement et demandent une bonne connaissance de la logique adoptée dans le but de guider et de contrôler le processus de preuve. Pour cette raison, d'autres approches bien connues et moins générales telles que les *systèmes de types* sont souvent considérées. Les systèmes de types ne visent pas à éviter des résultats inattendus mais à prévenir l'occurrence de termes dénués de sens par rapport à une spécification de types.

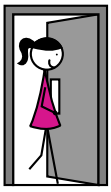
L'objectif général de cette thèse est de définir un tel système de types dans le cadre du développement du langage de programmation Tom [BBK+07, BBB+11]. Ce système de types est équipé de sous-typage pour le support de filtrage de motifs associatif sur des types de données algébriques avec des constructeurs variadiques.

Par la suite, nous présentons un aperçu des principaux sujets abordés dans cette thèse avant de souligner le contenu de chaque chapitre. Nous commençons par une brève discussion à propos de systèmes de types en tant qu'approche formelle pour la preuve de l'absence de termes mal-typés.

Systèmes de types

Les systèmes de types constituent un système d'inférence pour la classification de termes selon les genres (*i*.e. types) de valeurs calculés lors de leurs exécution. Une propriété souhaitable des systèmes de types est la sûreté : un programme classé comme bien-typé suite à la compilation ne doit pas provoquer d'erreurs d'exécution dues à des instructions « dénuées de sens ». Par exemple, la fonction de successeur de Peano suc prend un type nat en argument et renvoie un résultat du type nat comme l'indique son rang suc : nat → nat. Néanmoins, l'instruction suc(0) est considérée bien-typée tandis qu'une incompatibilités de types est détectée pour l'instruction suc(-1) puisque -1 n'est pas un nombre naturel.

Un système de types vu comme un système d'inférence est composé de règles



d'inférence (de types) dont l'application dépend de la vérification à effectuer :

- vérification de types, pour laquelle les règles de typage décrivent une approche déclarative pour vérifier statiquement si tous les termes d'un programme qui contiennent des annotations de type sont bien-typés,
- inférence de types, pour laquelle les règles de typage décrivent une approche algorithmique pour inférer statiquement le type de tous les termes d'un programme qui ne contiennent pas des annotations de type.

L'équivalence de ces deux approches peut être prouvée à travers la démonstration de deux propriétés :

- la correction, qui atteste que tout jugement de typage dérivable par les règles d'inférence de types est aussi dérivable par les règles de vérification de types,
- la complétude, qui assure qu'une solution validée par les règles de vérification de types peut être étendue à une solution proposée par les règles d'inférence de types.

John Mitchell [Mit84] définit l'inférence de types comme une forme de vérification de types. En général, les systèmes d'inférence de types sont basés sur des contraintes de types en suivant l'idée originale de Roger Hindley [Hin69]. Il a proposé un algorithme d'inférence de types pour générer et résoudre des systèmes de contraintes d'égalité. Ce travail a ete indépendamment développé plus tard par Robin Milner [Mil78] qui l'a étendu au support de l'abstraction d'un type de base par rapport à une variable de type. Il a suggéré la résolution des contraintes de types par *unification*, ce qui consiste à trouver une solution qui satisfait toutes les égalités. Milner a aussi prouvé la complétude et la correction du système de types. Le principal avantage de l'approche basée sur des contraintes est que l'ajout de nouveaux genres de contraintes ne nécessite que la modification des règles de typage et de l'algorithme de résolution de contraintes.

Dans cette thèse, nous nous intéressons à un système de types sûr pour des termes algébriques construits sur des opérateurs d'arité fixe et variable. Nous présentons une notion de type qui inclut des informations suffisantes, notamment pour la distinction des différents opérateurs variadiques. Nous nous concentrons sur un système de vérification de types ainsi que sur un système d'inférence de types basé sur des contraintes et équipé de *sous-typage*. En outre, nous souhaitons que celui-ci détecte au plus tôt un maximum d'erreurs de typage dans un programme.

Sous-typage

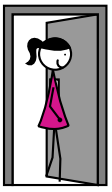
L'introduction de sous-typage dans les systèmes de types augmente leur expressivité puisque cela mène à une compatibilité de types plus permissive. Des généralisations du système de Hindley/Milner avec sous-typage ont été proposées par plusieurs auteurs, tels que Luca Cardelli [Car88] et John Mitchell [Mit84]. Ce dernier considérait le *sous-typage structurel*, où les relations de sous-typage sont déterminées structurellement en fonction des arités des constructeurs de types. Cardelli le voyait différemment : il a proposé la notion de *sous-typage*

nominal comme un ordre partiel sur des types. Cela implique l'ajout d'une nouvelle règle de typage pour garantir que si un type τ_1 est un sous-type de τ_2 alors chaque terme du type τ_1 est aussi un terme du type τ_2. Par conséquent, des programmes avec un bon comportement qui étaient auparavant considérés comme mal-typés peuvent désormais être acceptés.

En présence de sous-typage, l'inférence de types requiert l'extension du langage de contraintes à travers l'ajout de contraintes de sous-typage, en plus des contraintes d'égalité existentes. Or, au lieu de résoudre un ensemble d'équations, le problème de l'inférence de types revient à la simplification et la résolution d'un ensemble d'inéquations. Les systèmes de types basés sur des contraintes de sous-typage ont été profondément étudiés. Inspirés par le travail de Mitchell, You-Chin Fuh et Prateek Mishra [FM88] se sont concentrés sur le calcul des relations de sous-typage entre des types de base ainsi que sur la satisfiabilité de l'ensemble de contraintes. Cependant, l'idée de séparer le système d'inférence générique des détails concernant la simplification et la résolution de contraintes a été développée, en premier, par Stefan Kaes [Kae92] et, ensuite, par Valery Trifonov et Scott Smith [TS96].

Dans cette thèse, nous nous concentrons sur la définition tant d'un système de types équipé de sous-typage nominal que d'un algorithme de résolution de contraintes d'égalité et de sous-typage. En effet, certains auteurs ont abordé le problème de simplification de larges ensembles de contraintes. Alexander Aiken et Edward Wimmers [AW92] ont proposé un puissant mais aussi complexe algorithme d'inférence qui effectue la transformation des systèmes de contraintes tout en préservant l'ensemble de solutions. Outre Trifonov et Smith, François Pottier [Pot01] a aussi présenté une approche plus simple de simplification de contraintes en utilisant la notion d'*implication de contraintes*, *i*.e. la simplification de contraintes en préservant la solution. Son système de types supporte des types récursifs et tous les types de base sont organisés en un treillis. En outre, Pottier a décrit un algorithme de simplification de contraintes composé de plusieurs algorithmes complémentaires. Bien que traitant d'une notion de types différente de la nôtre, son approche a inspiré notre algorithme de résolution de contraintes, en particulier le sous-algorithme de *canonisation*. Toutefois, au-delà de la simplification de contraintes, nous nous concentrons sur la génération d'une seule solution en tant qu'ancrage de variables de types à des types de base.

Les règles de notre système d'inférence de types décrivent une façon algorithmique de mapper des types de termes d'une expression à un ensemble de contraintes. L'ensemble des contraintes obtenu est divisé en un ensemble de contraintes d'égalité et un ensemble de contraintes de sous-typage. Pour le premier ensemble, nous présentons une version non récursive de l'algorithme d'unification syntaxique employé par le système de types de Hindley/Milner. La propagation de contraintes de sous-typage est faite par la combinaison de trois phases de simplification. La phase de *simplification et clôture* élimine des contraintes de sous-typage réflexives et applique la clôture transitive sur des variables de types de l'ensemble de contraintes résultant. La *détection d'incom-*



patibilités cherche des contraintes de sous-typage qui ne sont pas valides par rapport à la spécification de types. Enfin, la *canonisation* assure que chaque variable de type apparaissant dans l'ensemble de contraintes ait au maximum un majorant et un minorant. La propagation de contraintes est combinée avec la résolution de contraintes. Les règles de *génération de solution* sont appliquées afin d'obtenir la solution avec les types les moins restrictifs. Ensuite, le *ramassage de miettes* détermine quelles sont les contraintes inutiles et les élimine. La terminaison de la totalité du processus de résolution de contraintes a aussi été prouvée.

Le système de types avec sous-typage conçu dans cette thèse est dédié au support de *filtrage de motifs associatif* sur des termes algébriques représentant des objets. En tant que caractéristique essentielle du paradigme orienté objet, l'héritage détermine des relations de sous-typage entre des objets qui sont vus comme des termes algébriques. Par conséquent, ces termes peuvent être des sous-types différents toujours manipulés de façon uniforme comme s'ils appartenaient à un supertype commun.

Filtrage de motifs associatif avec sous-typage

Les notions de types et de termes algébriques ne sont pas souvent présentes dans la majorité des langages orientés objet. De plus, la plupart de ces langages n'offrent pas de constructions de filtrage de motifs qui sont essentielles tant aux langages fonctionnels qu'aux langages basés sur la réécriture. Le filtrage de motifs est une opération destinée à verifier si un certain motif apparait dans les structures de données d'entrée. Si c'est le cas, alors les variables qui apparaissent dans le motif sont instanciées selon la solution de l'équation de filtrage et l'instruction pertinente est exécutée dans l'environnement étendu par cette solution.

De nos jours, une variété de langages disposent du filtrage de motifs, comme par exemple Maude [CDE+07], Scala [Ode11, Cre06], ELAN [BKK+98, BCD+06], ... L'un d'entre eux est Tom qui est le langage d'implémentation cible de cette thèse. Sa particularité est de permettre l'extension d'un langage hôte tel que Java par l'ajout de constructions de filtrage de motifs.

Nous pouvons utiliser Tom pour définir un type algébrique nat pour les entiers de Peano ainsi que leur opération d'addition :

```
nat = zero() | suc(n:nat)
...
public nat Plus(nat t1, nat t2) {
  %match(t1, t2) {
    zero(), x -> { return `x; }
    suc(y), x -> { return `suc(Plus(y,x)); }
  }
}
```

La construction **%match** implémente la notion de filtrage de motifs dans la méthode Java Plus. Cette méthode prend en argument deux termes t1 et t2 du

type algébrique nat représentant les entiers de Peano et renvoie leur somme qui est aussi du type nat :

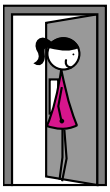
- si t1 = zero() et t2 = i pour un i donné, alors l'évaluation de la méthode Plus est t2, *i*.e. l'instance de l'objet représentant x,
- si t1 = suc(j) et t2 = i pour un i et un j donnés, alors l'évaluation de la méthode Plus est l'objet représentant le terme algébrique suc dont l'argument est la somme du sous-terme t1 avec t2.

Le filtrage de motifs permet la discrimination des constructeurs algébriques et la déconstruction des termes algébriques en leurs sous-termes. Cela constitue une opération essentielle pour l'évaluation des termes à travers la *réécriture de termes*. La réécriture de termes est un modèle de calcul pour le remplacement de sous-termes de termes algébriques avec d'autres termes. Ce modèle est souvent combiné avec des théories équationelles comme proposé dans les travaux de Gerald Peterson et Mark Stickel [PS81], qui plus tard a été généralisé par Jean-Pierre Jouannaud et Hélène Kirchner [JK86]. Le langage Tom étant un langage de réécriture, il fournit du filtrage de motifs modulo associativité pour les termes construits sur des opérateurs variadiques. Par exemple, nous définissons un opérateur variadique concNat pour le type algébrique nat représentant des listes d'entiers de Peano et calculons leur somme :

```
nat = zero() | suc(n:nat) | concNat(nat*)
...
public nat ListPlus(nat t) {
  %match(t) {
    concNat()        -> { return `zero(); }
    concNat(x)       -> { return `x; }
    concNat(x,y,z*)  -> { return(listPlus(concNat(Plus(`x,`y),z*)); }
  }
}
```

Lorsque l'on considère des listes comme des motifs, Tom applique le filtrage de motifs modulo associativité et élément neutre sur ces listes par l'utilisation de variables étoiles telles que z* figurant parmi les arguments de la liste. En effet, z* représente une sous-liste contiguë de t dont le symbole de tête est le même que celui de la liste dans laquelle il apparait, *i*.e. concNat. Les variables x et y à leur tour représentent les entiers de Peano dont les symboles de tête sont différents de concNat. Cependant, grâce à la théorie de l'associativité et de l'élément neutre, le motif concNat(x,y,z*) filtre concNat(zero(),concNat(zero(),suc(zero())) indicant concNat(suc(zero())) comme l'instance de z et zero() comme l'instance de x et de y.

Dans cette thèse, nous nous intéressons à l'amélioration de l'expressivité du filtrage de motifs fourni par Tom en tirant parti des différents niveaux de hiérarchie de types. L'évaluation de l'équation de filtrage dont le motif n'est pas une variable consiste effectivement à vérifier que le motif et le terme à filtrer ont tous les deux le même symbole de tête. Néanmoins, la présence de relations de sous-typage entre des types algébriques requiert de vérifier que le terme à filtrer *se réduit* à un terme dont le type est un sous-type de celui du motif. Une suite



de réductions constitue l'évaluation des termes et des expressions du langage. Nous décrivons l'évaluation du code Tom à travers la définition de la sémantique opérationnelle à grand pas du langage Tom selon le formalisme définit par Gilles Kahn [Kah87]. La relation de réduction de termes et expressions Tom est inspirée de celle du langage OCaml présentée par Didier Rémy [Rém02]. Toutefois, l'évaluation des expressions Tom concernant le filtrage de motifs avec sous-typage est plutôt influencée par les règles de simplification décrites par Delia Kesner [Kes91] pour la compilation du filtrage de motifs dans des langages ordo-sortés. La formalisation de la sémantique opérationnelle de Tom est essentielle pour démontrer la sûreté de notre système de types. Par conséquent, nous prouvons que toute expression Tom bien-typée est un terme Tom ou se réduit à un terme Tom (propriété de *progrès*). Nous prouvons aussi que chaque pas d'évaluation préserve le type de l'expression Tom (propriété de *préservation* ou de *réduction du sujet*).

Le langage de programmation Tom se caractérise par le support de l'analyse et de la transformation de plusieurs genres de structures arborescentes. Pour aider le programmeur à spécifier et à implémenter des outils d'analyse et de transformation, en plus de signatures de termes du premier ordre et filtrage de motifs, Tom fournit d'autres concepts et constructions de haut niveau telles que des règles de réécriture et des stratégies de contrôle pour l'application de ces règles. Le sujet de cette thèse concerne l'étude des algorithmes de typage pour les règles et les stratégies du langage Tom. Nous nous concentrons particulièrement sur le support de la définition de types algébriques équipés de sous-typage. Nous avons pour objectif de proposer un meilleur langage Tom qui fournit à la fois une plus forte expressivité et plus de sûreté afin d'assurer que les transformations décrites par des règles de réécriture préservent le type des termes. La possibilité de concevoir des compilateurs plus sûrs en utilisant Tom a un impact significatif sur la prévention de la génération de termes mal-typés incompatibles avec leurs signatures de type.

Contributions

Cette thèse peut être considérée comme une contribution pour le système de types de Tom en trois étapes, où chacune d'entre elles construit et améliore le résultat précédent afin d'augmenter la sûreté et l'expressivité du langage. Évidemment, tout le développement théorique a été fait en gardant à l'esprit un objectif pratique.

Un système de types pour Tom

Au début de cette thèse, Tom était capable d'effectuer des tâches (très) simples de typage : par exemple, il pouvait inférer les types des variables apparaissant dans les conditions des règles de réécriture et ensuite les propager, le cas échéant, dans les occurrences de ces variables dans le côté droit des règles.

Cependant, en absence d'un système de types formellement défini, la phase de compilation concernant le typage (*i*.e. la vérification de types ainsi que l'inférence de types) n'était pas bien délimitée. Par conséquent, la phase de typage effectuait des tâches (telles que le désucrage syntaxique) liées à d'autres phases qui effectuaient à leur tour des tâches de typage redondantes.

La première contribution de cette thèse est donc de présenter un système de types formalisé pour les termes algébriques de Tom afin d'obtenir plus de sûreté sur des types. Nous proposons ensuite un système de vérification de types pour examiner si tous les termes algébriques *purs* d'un programme Tom sont bien-typés. Nous entendons par *pur* les termes algébriques qui ne sont pas construits sur des types concrets de données. En outre, nous proposons un système d'inférence de types pour inférer des types des variables apparaissant dans des termes algébriques (même dans ceux qui ne sont pas purs). Cela mène à un algorithme d'inférence capable de simultanément remplir l'information de type des termes algébriques ainsi que de les vérifier. Ce type de vérification est possible puisque les types infér'es et les annotations de type sont utilisés dans la génération et la résolution des contraintes de types.

L'implémentation du typeur en utilisant des règles définies pour l'inférence de types conduit naturellement à plusieurs modifications au niveau des développeurs. La représentation interne du code de Tom doit être capable de manipuler des contraintes de types à travers la définition de leurs constructeurs de données. L'effet de bord de cette première implémentation est un nettoyage général de code (refactorisation de code, revue de l'architecture de code, etc).

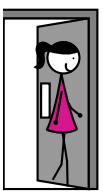
Tant la formalisation que l'implémentation du système de types en entier sont présentées en détail dans le Chapitre 3.

Sous-typage

La prochaine contribution de cette thèse est d'ajouter les fonctionnalités de sous-typage au système de types susmentionné en vue d'augmenter l'expressivité des constructions de filtrage de motifs fournies par Tom, tout en gardant la sûreté obtenue. Cette étape est naturellement axée sur le système résultant de la première contribution et bénéficie de différents niveaux de hiérarchie de classes fournie par le langage hôte Java.

La solution retenue ici est donc celle d'enrichir le système de types formel de Tom avec sous-typage comme un ordre partiel sur les types. Cela inclut l'ajout d'une nouvelle règle au système de types formalisé pour attester que si un type s_1 est un sous-type de s_2 alors tout terme du type s_1 est aussi du type s_2. Par conséquent, l'ajout de sous-typage au système d'inférence de types basé sur des contraintes implique d'étendre le langage de contraintes avec des contraintes de sous-typage, en plus de celles d'égalité. En outre, le problème de l'inférence de types est réduit à la résolution d'équations et d'inéquations.

L'implémentation de ce nouveau système avec sous-typage a des conséquences pour les utilisateurs et les développeurs. Au niveau de l'utilisateur, elle nécessite une construction supplémentaire pour la déclaration des relations de sous-



typage : la construction **extends** est combinée avec **%typeterm** durant la définition des constructeurs de types. Cela permet à l'utilisateur de spécifier la relation d'héritage entre des types algébriques lors de la définition d'un ancrage à la main. Ainsi, la hiérarchie de types manipulant des types algébriques devrait respecter celle manipulant leur types concrets respectifs. En interne, cela mène à la modification de la représentation du langage afin de manipuler plus d'annotations de type et des relations de type, ainsi que la définition de constructeurs spécifiques de données pour les contraintes de sous-typage. Cela permet l'implémentation d'un nouvel algorithme de simplification et de résolution des contraintes de sous-typage.

Dans le Chapitre 4, nous décrivons formellement les règles de typage et l'algorithme de résolution de contraintes ainsi que les aspects d'implémentation.

Sémantique Opérationelle

Enfin, l'intégration de sous-typage dans Tom influe également sur les algorithmes de filtrage implémentés dans le compilateur. Notre troisième contribution est la redéfinition de ces algorithmes de filtrage.

En présence des types partiellement ordonnés par sous-typage, le filtrage de motifs consiste en deux genres de vérification :

- égalité (même celle modulo une théorie équationelle) des constructeurs de type du motif et du sujet correspondant au filtrage structurel tel qu'effectué avec des types non ordonnés,
- constatation que le sujet se réduit à un terme du type s_1 qui est un sous-type du type s_2 du motif.

Ainsi, les règles de réduction qui composent les algorithmes de filtrage syntaxique, de filtrage modulo $\mathcal{A}$ et de filtrage modulo $\mathcal{AU}$ doivent aussi considérer des contraintes de sous-typage. À cet effet, dans le Chapitre 3, nous définissons la syntaxe de base de Tom ainsi que sa sémantique opérationnelle qui décrit l'évaluation de code Tom. Ensuite, dans le Chapitre 4, nous adaptons les règles de réduction afin de permettre l'inclusion de types au niveau des motifs. Étant donné que le système Tom n'analyse pas le code hôte, le code responsable de la vérification des solutions d'une équation de filtrage doit être traduit en instructions sémantiquement équivalentes du langage hôte Java. Dans le Chapitre 5, le résultat de l'intégration de sous-typage dans Tom a été testé dans un étude de cas qui utilise Tom pour la transformation de modèles.

Le langage Tom dans le contexte de la réécriture de termes

Le langage Tom est développé dans l'équipe projet Pareo [1] depuis 2001. Étant un langage dédié à un domaine spécifique (DSL) embarqué dans Java, Tom

1. Reférence disponible à http ://pareo.loria.fr . Dernière visite : Février 2012.

fournit des constructions à base de règles et de filtrage de motifs permettant l'encodage de règles de réécriture dans du code source orienté objet.

Bien qu'étant un langage embarqué, les constructions de filtrage de motifs fournies par Tom peuvent être appliquées sur des termes algébriques, indépendamment de leurs représentations concrètes. En ce sens, on doit spécifier les types de données algébriques (et abstraites) manipulés par Tom, les types de données concrets utilisés pour les représenter dans le langage hôte ainsi qu'un *ancrage* définissant la façon dont se fait cette représentation. Selon cet ancrage, le compilateur transforme les applications de constructions de filtrage de motifs en des manipulations des types de données concrets. Un ancrage est composé des déclarations des types de données algébriques et de leurs implémentations respectives. Des constructeurs de type sont définis par la construction Tom **%typeterm**, alors que les constructeurs de données sont définis par les constructions Tom **%op** et **%oplist**. Comme pour les opérateurs syntaxiques, les opérateurs variadiques d'un type algébrique sont définis par la construction **%oplist** où les termes algébriques construits à partir des opérateurs variadiques sont souvent dénommés *listes*. Lors de la définition d'une liste, l'utilisateur doit spécifier comment construire et déconstruire une liste, *i*.e. la façon de construire une liste vide et d'insérer un élément dans une liste ainsi que la façon d'obtenir à la fois le premier élément et le reste d'une liste.

Nous définissons maintenant un ancrage entre un type de donnée algébrique et les classes Java qui l'implémentent. Nous commençons par définir les constructeurs de type pour les types algébriques Nat et NatList.

```
1  %typeterm Nat {
2    implement    { JNat }
3    is_sort(s)   { (s instanceof JNat) }
4    equals(t1,t2) { (t1.equals(t2)) }
5  }

7  %typeterm NatList {
8    implement    { JNatList }
9    is_sort(s)   { (s instanceof JNatList) }
10   equals(t1,t2) { (t1.equals(t2)) }
11 }
```

Les types algébriques Nat et NatList sont respectivement implémentés par les types concrets JNat et JNatList. Les types algébriques ne sont connus que par Tom, puisqu'ils ne sont pas déclarés dans le programme Java résultant. Les rangs des opérateurs d'un type algébrique sont définis par la construction **%op** qui permet de spécifier la façon de construire et de déconstruire des termes algébriques avec ces opérateurs. Nous définissons les opérateurs zero et suc pour le type Nat et la liste concNat pour le type NatList comme suit :

```
1  %op Nat zero() {
2    is_fsym(s) { (s instanceof Jzero) }
3    make()     { new Jzero() }
4  }
```

```
6  %op Nat suc(n:Nat) {
7    is_fsym(s)    { (s instanceof Jsuc) }
8    get_slot(n,s) { ((Jsuc)s).n }
9    make(t0)      { new Jsuc(t0) }
10 }

12 %oplist NatList concNat(Nat*) {
13   is_fsym(s)         { (s instanceof JconcJNat) }
14   get_head(l)        { ((JconcJNat)l).head }
15   get_tail(l)        { ((JconcJNat)l). tail  }
16   is_empty(l)        { ((JconcJNat)l).isEmpty() }
17   make_empty()       { new JconcJNat() }
18   make_insert(t,l)   { new JconcJNat(t,l) }
19 }
```

La définition d'ancrages entre les types de données algébriques Tom et les types de données concrets Java permet à Tom de manipuler des termes Java comme des termes Tom. Pour une utilisation standard de Tom, ces ancrages peuvent être automatiquement générés par l'outil Gom écrit par Antoine Reilles et présenté en détail dans son travail [Rei07]. À partir d'une signature algébrique multi-sortée, Gom génère à la fois un ancrage pour les types de données algébriques décrites par la signature et un ensemble de classes Java qui l'implémentent.

Exemple 1. *L'ancrage écrit manuellement et présenté jusqu'ici peut être généré par Gom à partir de la signature algébrique suivante :*

```
1    module ExampleGom
2    abstract syntax

4    Nat = zero()
5        | suc(n:Nat)

7    NatList = concNat(Nat*)
```

Les programmes Tom peuvent vérifier si un motif donné apparait dans des structures de données du langage hôte et instancier des variables selon la solution de l'équation de filtrage.

Exemple 2. *Considérons l'ancrage précédent écrit manuellement. L'addition de deux entiers de Peano peut être codée comme suit :*

```
1  public JNat peanoPlus(JNat t1, JNat t2) {
2    %match(Nat t1, Nat t2) {
3      zero(), x -> { return `x; }
4      suc(y), x -> { return `suc(peanoPlus(y,x)); }
5    }
6    return null;
7  }
```

La méthode Java peanoPlus prend deux termes t1 et t2 du type JNat représentant les entiers de Peano et retourne leur somme qui est aussi du type JNat. L'exemple utilise la construction ‘ (backquote) pour construire des termes algébriques, permettant la manipulation de variables x et y dans l'instruction Java.

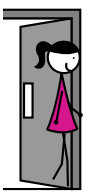
Le langage Tom fournit du filtrage de motifs modulo $\mathcal{AU}$ pour les motifs composés de listes (*i*.e. des opérateurs variadiques). Le filtrage modulo $\mathcal{AU}$ est dénommé *filtrage de listes* et constitue une instance du filtrage modulo une théorie équationelle. Lorsque l'on considère des listes comme des motifs, la notion de filtrage de listes peut leur être appliquée par l'utilisation de variables annotées avec *. Ces variables sont appelées *variables étoiles* et apparaissent dans les arguments des listes. Les variables étoiles diffèrent des variables simples parce qu'elles représentent une sous-liste contiguë du sujet. Cela indique une variable étoile représentant une sous-liste dont le symbole de tête est le même de celui de la liste dans laquelle la variable apparait. Par ailleurs, pour un opérateur variadique donné, l'élément neutre est la liste vide construite à partir de cet opérateur. Par conséquent, si nous considérons l'équation de filtrage

$$\mathsf{concNat(x*,y*)} \ll^{?} \mathsf{concNat(zero(),suc(zero(),suc(suc(zero()))))}$$

l'opérateur de concaténation :: pour les entiers de Peano et les listes de nombres entiers de Peano est une interprétation possible pour concNat. Ainsi, les entiers de Peano sont associés à des constructeurs de données du type Nat. Cela indique que concNat(x*,y*) représente $(x :: y)$ et concNat(zero(),suc(zero(),suc(suc(zero())))) représente $(0 :: 1 :: 2)$. En outre, la liste vide concNat() représente un élément neutre () pour l'opérateur de concaténation.

Par souci d'expressivité, le langage Tom fournit une syntaxe supplémentaire de la construction **%match** qui devient une liste de paires (condition, action). Chaque condition correspond à une ou plusieurs conditions combinées par les connecteurs booléens && et || représentant respectivement les opérateurs de conjonction et de disjonction. Les conditions de filtrage prennent la forme motif << sujet (ou la forme motif << typeMotif sujet si on souhaite expliciter les types) exprimant des conditions affirmatives. Des conditions négatives peuvent être déclarées par la notion de complément de motifs introduite par le symbole !. Le symbole de complément ! peut apparaître dans n'importe quelle position du motif et autant que de fois que nécessaire, ce qui constitue, par conséquent, un anti-motif. En outre, les conditions numériques sont également supportées par Tom permettant la comparaison de termes par l'utilisation des opérateurs < (inférieur à), <= (inférieur ou égal à), > (supérieur à), >= (supérieur ou égal à), == (égal à) ou != (différent de).

Un alias peut être associé aux sous-termes des motifs par l'utilisation de la construction @. En résolvant une condition de filtrage (ou d'anti-filtrage), la variable représentant un alias est instanciée par l'instance du sous-terme correspondant du sujet.



Lors de la définition des signatures algébriques d'opérateurs non-variadiques, chacun de leurs arguments est nommé. Cela permet de spécifier un certain sous-terme du motif construit à partir d'un opérateur non-variadique en faisant référence uniquement à son nom. La construction [ ] permet à la condition de filtrage de considérer des arguments spécifiques et d'ignorer ceux restants.

La construction **%match** permet à la fois la déconstruction des termes algébriques et la définition des règles de réécriture. Dans Tom, l'application des règles de réécriture peut être contrôlée par l'utilisation de *stratégies* définissant quelles règles doivent être appliquées dans quelles positions des termes. Le langage de stratégies proposé par Tom est constitué d'une bibliothèque Java de combinateurs de stratégies inspirée par le langage Stratego [VBT98] qui à son tour était inspirée par le langage de stratégie d'ELAN.

Les stratégies élémentaires sont créées par la construction **%strategy** et correspondent à un système de réécriture de termes (TRS) représenté par un objet Java objet de la classe Strategy. Les règles du TRS ne sont appliquées qu'une seule fois sur la position de tête d'un terme donné.

Le système de types de Tom

La notion de *type* utilisée pour la classification des termes d'un langage est nécessaire pour parler rigoureusement des systèmes de types et de leurs propriétés. Nous devons également présenter formellement la syntaxe de base de Tom qui nous intéresse. L'évaluation des expressions construites à partir de cette syntaxe est décrite par la suite par une sémantique opérationnelle. Ensuite, nous décrivons notre système de vérification de types comme un calcul de preuve pour des jugements de typage sur des expressions Tom annotées avec des types. Nous définissons un système d'inférence de types pour les expressions construites à partir de variables n'ayant aucune information de type. Cela est décrit comme un algorithme pour calculer les annotations de type non spécifiées. La formalisation résultante nous permet de raisonner sur les deux aspects de la logique de langage de programmation : la syntaxe et la sémantique.

Termes et types Tom

Comme on le voit dans la section précédente, les termes algébriques Tom sont construits à partir de variables, d'opérateurs syntaxiques et d'opérateurs variadiques. Nous introduisons respectivement les ensembles $\mathcal{V}$, $\mathcal{F}$ et $\mathcal{F}^\star$ pour les représenter. Puisque les termes Tom sont construits à partir de signatures variadiques multi-sortées, nous présentons un ensemble $\mathcal{S}$ de sortes de base représentées par des types de données algébriques Tom. À ce stade, nous pouvons encore construire des termes à partir d'opérateurs syntaxiques et variadiques d'une même sorte. Toutefois, nous devons distinguer ces deux genres de termes au cours de la vérification de types et de l'inférence de types. Par exemple, considérons un opérateur variadique $\mathsf{v} \in \mathcal{F}^\star$, deux variables $\mathsf{x}, \mathsf{y} \in \mathcal{V}$ et un opérateur g

$\in \mathcal{F} \cup \mathcal{F}^\star$. L'insertion de g(x) dans une liste g(x) peut conduire à deux différents résultats en fonction de la déclaration de g(x) :

1. si $\mathsf{g} \in \mathcal{F}^\star$ et $\mathsf{g} = \mathsf{v}$, alors, par aplatissement, le résultat est la concaténation des deux listes, i.e. v(x,y) ;
2. si $\mathsf{g} \in \mathcal{F}^\star$ et $\mathsf{g} \neq \mathsf{v}$ ou si $\mathsf{g} \in \mathcal{F}$, alors le résultat est l'insertion d'un élément dans une liste, i.e. v(g(x),y).

Par conséquent, nous nous intéressons à la distinction entre les listes avec le même symbole de tête et les termes restants. Nous introduisons la notion de sorte décorée avec des symboles de fonction pour la classification de termes, par exemple s^g avec $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^\star$. Nous définissons aussi un symbole spécial ? pour les cas où la décoration n'a pas d'importance, par exemple, lors de la spécification du domaine d'un symbole de fonction. Les sortes décorées sont aussi dénommées *types clos*, car ce sont les types les plus simples avec lesquels un terme peut être classé. Nous introduisons un type clos spécial wt pour classer des expressions *bien-typées* qui ne sont pas des termes, comme par exemple les conditions de filtrage et les constructions backquote (`) entre autres. Finalement, l'ensemble de types $\mathcal{T}y$ est composé de termes non typés construits à partir d'un ensemble de *variables de type*, de $\{wt\}$ et de l'ensemble des sortes décorées, *i.*e. la combinaison de $\mathcal{S}$ et $\mathcal{F} \cup \mathcal{F}^\star \cup \{?\}$.

Définition 3 (Types)**.** *Soit wt une constante spéciale et $\mathcal{X}$ un ensemble infini dénombrable de* variables de type *notées α, β, etc. L'ensemble de types $\mathcal{T}y$ est constituée des termes non sortés définis par la grammaire algébrique suivante :*

$$\tau ::= \alpha \mid s^h \mid wt$$

où $\tau \in \mathcal{T}y$, $\alpha \in \mathcal{X}$, $s \in \mathcal{S}$, $h \in \mathcal{F} \cup \mathcal{F}^\star \cup \{?\}$.

Un type clos *est un type τ sans variables, i.e. $\mathcal{V}ar(\tau) = \emptyset$.*

L'égalité des sortes décorées consiste en l'égalité syntaxique des sortes enrichie par une comparaison particulière entre ses décorations.

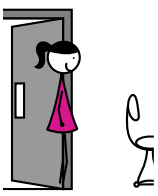
Définition 4 (Égalité des sortes décorées)**.** *Considérons l'ensemble de types $\mathcal{T}y$, i.e. des sortes décorées.* L'égalité des sortes décorées, *notée $=_t$, est définie par l'égalité modulo la propriété suivante :*

$$\forall s_1^{h_1}, s_2^{h_2} \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\}), (s_1^{h_1} =_t s_2^{h_2} \Leftrightarrow (s_1 = s_2 \wedge (h_1 = h_2 \vee h_1 \ = ? \vee h_2 \ = ?)))$$

Nous écrivons $s_1^{h_1} \neq_t s_2^{h_2}$ pour signifier que $\neg(s_1^{h_1} =_t s_2^{h_2})$.

Or, nous étendons la notion de *sous-typage* sur l'ensemble de types $\mathcal{T}y$ composé de sortes décorées. Cela mène à $\mathcal{T}y$ équipé d'un autre ordre partiel $<:_t$.

Définition 5 (Sous-typage sur de sortes décorées)**.** *Considérons l'ensemble de types $\mathcal{T}y$, i.e. de sortes décorées. Le* sous-typage sur des sortes décorées, *notée $<:_t$, est un ordre partiel sur $\mathcal{T}y$ défini par sous-typage modulo la propriété suivante :*



$$\forall s_1^{h_1}, s_2^{h_2} \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\}), (s_1^{h_1} <:_t s_2^{h_2} \Leftrightarrow (s_1 <: s_2 \wedge (h_1 = h_2 \vee h_2 \ = ?)))$$

Dans notre système de types, les types sont interprétés comme des termes non sortés et le sous-typage est nominal. Ainsi, des relations de sous-typage doivent être explicitement déclarées, bien que l'héritage multiple et la surcharge d'opérateurs soient interdits. Par exemple, étant donné les types $\mathsf{Neg}^?$, $\mathsf{ZNat}^?$, $\mathsf{Nat}^?$ et $\mathsf{Int}^?$, le système de type accepte la déclaration $\mathsf{Neg}^? <:_t \mathsf{Int}^? \wedge \mathsf{Nat}^? <:_t \mathsf{Int}^?$, mais, dans ce cas, il refuse la déclaration $\mathsf{ZNat}^? <:_t \mathsf{Neg}^? \wedge \mathsf{ZNat}^? <:_t \mathsf{Nat}^?$. En outre, dans Tom, un symbole de fonction donné ne peut pas être surchargé par deux codomaines de types différents. Par exemple, le symbole de fonction suc ne peut pas avoir deux codomaines de types $\mathsf{Neg}^?$ et $\mathsf{Nat}^?$.

Étant donné l'ensemble de types $\mathcal{T}y$, l'ordre partiel $<:_t$ sur $\mathcal{T}y$, les ensembles $\mathcal{F}$ et $\mathcal{F}^\star$ de symboles de fonction et un ensemble infini dénombrable de variables $\mathcal{V}$, les termes Tom sont construits ci-après à partir de l'algèbre de termes ordo-sortée $\mathcal{T}(\mathcal{T}y, <:_t, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$. L'ensemble de termes clos de Tom est désigné par $\mathcal{T}(\mathcal{T}y, <:_t, \mathcal{F} \cup \mathcal{F}^\star)$.

Comme nous l'avons vu précédemment, les types de données algébriques dans Tom sont implémentés par des types de données concrets dans Java et cette correspondance est définie par un ancrage. Ainsi, la déclaration des relations de sous-typage doit être faite à la fois dans les constructeurs de type et dans les classes Java qui les implémentent. Nous montrons ci-dessous comment faire cela dans un extrait de code qui étend l'ancrage écrit manuellement et présenté antérieurement. Nous définissons deux nouveaux constructeurs de types ZNat et Int dont les constructeurs de données sont respectivement zero et uminus :

```
1  static class JInt{ }
2  static class JNat extends JInt { }
3  static class JZNat extends JNat { }

5  static class Juminus extends JInt {
6    public JNat n;
7    public Juminus() { }
8    public Juminus(JNat n) { this.n = n; }
9  }

11 static class Jzero extends JNat {
12   public Jzero() {}
13 }

15 %typeterm Int {
16   implement    { JInt }
17   is_sort(s)   { (s instanceof JInt) }
18   equals(t1,t2) { (t1.equals(t2)) }
19 }

21 %typeterm Nat extends Int {
22   implement    { JNat }
23   is_sort(s)   { (s instanceof JNat) }
24   equals(t1,t2) { (t1.equals(t2)) }
25 }
```

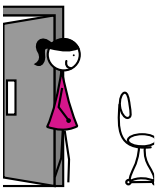
```
27  %typeterm ZNat extends Nat {
28    implement    { JZNat }
29    is_sort(s)   { (s instanceof JZNat) }
30    equals(t1,t2) { (t1.equals(t2)) }
31  }

33  %op Int uminus(n:Nat) {
34    is_fsym(s)   { (s instanceof Juminus) }
35    get_slot(n,s) { ((Juminus)s).n }
36    make(t0)     { new Juminus(t0) }
37  }

39  %op ZNat zero() {
40    is_fsym(s) { (s instanceof Jzero) }
41    make()     { new Jzero() }
42  }
```

La relation entre les types ZNat, Nat et Int est déclarée par le mot-clé **extends** de telle sorte que ZNat est un sous-type de Nat qui est un sous-type de Int : lignes 2 et 3, pour les types Java ; lignes 21 et 27, pour les types algébriques. Ainsi, le système de types interprète cette information comme $\mathsf{ZNat}^? <:_t \mathsf{Nat}^? \wedge \mathsf{Nat}^? <:_t \mathsf{Int}^?$. Cette fois-ci, le constructeur de donnée zero est redéfini pour ZNat (au lieu de Nat) à la ligne 39 et une classe Java et un nouveau constructeur de donnée uminus pour Int sont déclarés respectivement aux lignes 5 et 33. Cette nouvelle hiérarchie de types est considérée par Tom lors de l'exécution du filtrage de motifs afin d'accepter des motifs dont les types sont soit identiques, soit des sous-types de ceux des sujets. En conséquence, nous notons que la méthode peanoPlus de l'exemple 2 fonctionne toujours, même après la redéfinition du type de zero. Nous illustrons l'utilisation du filtrage de motifs sur des termes ordo-sortés à travers un exemple considérant des motifs des types ZNat, Nat and Int.

Exemple 6. *Considérons l'ancrage défini précédemment. La méthode Java suivante décrit un algorithme pour calculer l'addition de deux entiers (éventuellement négatifs).*

```
43  public JInt add(JInt t1, JInt t2) {
44    %match {
45      zero()     << t1 && x              << t2     -> { return `x; }
46      y          << t1 && uminus(zero()) << t2     -> { return `y; }
47      suc(y)     << t1 && x              << Nat t2 -> { return `suc(peanoPlus(y,x)); }
48      suc(y)     << t1 && uminus(suc(x)) << t2     -> { return `add(y,uminus(x)); }
49      uminus(y) << t1 && uminus(x)     << t2     -> { return `uminus(peanoPlus(y,x)); }
50      uminus(y) << t1 && suc(x)        << t2     -> { return `add(suc(x),uminus(y)); }
51    }
52    return null;
53  }
```

Nous notons que le paramètre de type de la condition de filtrage à la ligne 47 est nécessaire pour permettre à x d'être passé en tant qu'argument de peanoPlus dans le côté droit de la règle.



La notion de type est vraiment utile pour le typage des expressions Tom. Or, nous définissons une correspondance appelé *nettoyage de décoration* entre les types et les sortes de base. Cela permet aux types d'être convertis en des sortes non décorées. Nous utilisons cette correspondance principalement lorsque nous représentons les annotations de type explicites de la construction **%match** à travers un type.

Définition 7 (Nettoyage de décoration)**.** *La fonction* $|\cdot|$ *est une fonction partielle de* $\mathcal{T}y$ *dans* $\mathcal{S}$ *définie par :*

$$|s^h| = s$$

où $s \in \mathcal{S}$ *et* $h \in \mathcal{F} \cup \mathcal{F}^\star \cup \{?\}$.

Étant donné l'ensemble de types $\mathcal{T}y$, les ensembles de symboles de fonction $\mathcal{F}$ et $\mathcal{F}^\star$ et un ensemble infini dénombrable de variables $\mathcal{V}$, les termes Tom sont construits à partir de l'algèbre de termes $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$. L'ensemble de termes clos de Tom est désigné par $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$.

Définition 8 (Termes Tom)**.** *Le langage de termes Tom est spécifié par la grammaire algébrique suivante :*

$$term ::= \mathsf{x} \mid \mathsf{x*} \mid \mathsf{f}(\underbrace{term, \ldots, term}_{n}) \mid \mathsf{g}(term,\ldots,term)$$

où $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}_n$ *et* $\mathsf{g} \in \mathcal{F}^\star$.

Un code Tom peut contenir du code Java qui peut à son tour contenir du code Tom et ainsi de suite. Puisque nous envisageons de vérifier et d'inférer les types des termes Tom, nous considérons une abstraction consistant uniquement des expressions composées de termes Tom. Ainsi, les variables Java liées aux termes Tom sont représentées par de variables de $\mathcal{V}$. En outre, les fonctions Java qui manipulent de termes Tom ou qui sont manipulées par des termes Tom sont entièrement représentées par des termes Tom. Les objets Java restants sont rejetés.

Définition 9 (syntaxe abstraite de base de Tom)**.** *La* syntaxe abstraite de base de Tom *est spécifiée par la grammaire algébrique suivante :*

$$\begin{array}{lcl}
code & ::= & \{rule;\ldots;rule\} \mid \text{`}term \mid \mathsf{x} := \text{`}term \\
rule & ::= & cond \longrightarrow action \\
cond & ::= & pattern \ll |\tau|\ term \mid term \diamond term \mid cond \wedge cond \mid cond \vee cond \\
pattern & ::= & \mathsf{x} \mid \mathsf{x*} \mid \mathsf{g}(pattern,\ldots,pattern) \mid \mathsf{x}@pattern \mid !pattern \\
term & ::= & \mathsf{x} \mid \mathsf{x*} \mid \mathsf{g}(term,\ldots,term) \\
action & ::= & code \mid (action;\ldots;action)
\end{array}$$

où $\mathsf{x} \in \mathcal{V}$, $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^\star$ *et* $\tau \in \mathcal{T}y \backslash \{wt\}$.

Les termes construits à partir de cette grammaire sont nommés des expressions Tom.

Un code Tom est un ensemble de code.

Un *code* peut être une instruction d'affectation, un terme avec un backquote ou une liste de règles *cond* $\longrightarrow$ *action*. L'affectation d'un terme t à une variable x est représentée par x := `t. Le côté gauche d'une règle est soit une condition simple, soit une conjonction/disjonction de conditions simples. Une condition simple est soit une *condition de filtrage*, soit une *condition numérique*. Les *conditions de filtrage* sont des équations de filtrage $t_1 \ll^? |\tau| t_2$ où le symbole « ? » est omis par des raisons de lisibilité. τ désigne un type et $|\tau|$ est sa forme non décorée (c.f. Définition 7). Nous appelons *anti-motif* les motifs composés d'un ou plusieurs !*pattern*. Les *conditions numériques* sont les conditions de la forme $t_1 \diamond t_2$ où $\diamond$ signifie la représentation des opérateurs logiques $=$, $\neq$, $<$, $\leq$, $>$ et $\geq$. Le côté droit d'une règle peut être composé d'un ou plusieurs *code*. Ainsi, une action peut être une séquence d'actions, *i*.e. une séquence de listes contenant de règles et/ou de termes avec un backquote. Un `*term* correspond à la construction backquote permettant à un terme Tom d'être intégré dans un bloc de code Java. En outre, des variables anonymes _ n'apparaissent pas dans la grammaire, car elles sont représentées par de *variables fraîches* dont le nom n'est pas utilisé ailleurs.

Exemple 10. *Considérons l'ancrage écrit manuellement dans la section précédente et la méthode Java* printPositiveIntegers *suivante :*

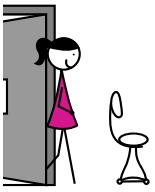
```
1  public void printPositiveIntegers (JNatList nList) {
2    int counter = 0;
3    %match {
4      concNat(x*,pnat@suc(y),z*) << NatList nList -> {
5        counter++;
6        System.out.println("Positive integer #" + counter + " = " + `pnat);
7      }
8    }
9  }
```

La construction **%match** *peut être représentée par le code Tom suivant :*

$$\{\text{concNat(x*,pnat@suc(y),z*)} \ll |NatList^?| \text{ nList} \longrightarrow \text{`pnat}\}$$

Sémantique opérationnelle

Nous décrivons brièvement la sémantique opérationnelle à grand pas du langage Tom selon le formalisme défini par Gilles Kahn [Kah87]. Le travail de Didier Rémy [RÓ2] a également inspiré la description syntaxique de l'évaluation du code Tom qui est définie par une relation de réduction sur les expressions, dénotée par $e \Downarrow e'$. Nous introduisons la notion *d'environnement* pour enregistrer l'état d'évaluation d'un code, *i*.e. les *valeurs* de toutes ses variables courantes.



Définition 11 (Valeurs). *L'ensemble des* valeurs $\mathcal{V}al$ *est composé d'un ensemble de termes clos* $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ *avec l'ajout de deux constantes distinctes* accept *et* reject *:*

$$\dot{v} ::= \dot{u} \mid \mathsf{accept} \mid \mathsf{reject}$$
$$\dot{u} ::= \mathsf{g}(\dot{u}, \ldots, \dot{u})$$

où $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^\star$, $\dot{u} \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ *et* $\dot{v} \in \mathcal{V}al$.

Nous notons $\flat(\dot{u})$ *l'opération d'aplatissement appliquée sur les valeurs.*

Définition 12 (Environnement). *Un* environnement *(d'évaluation)* ρ *est une fonction partielle de* $\mathcal{V}$ *dans* $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$. *Soit* $\mathsf{x} \in \mathcal{V}$ *et* $\mathsf{t} \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$. *Nous notons* $\rho[\mathsf{x} \mapsto \mathsf{t}]$ *l'environnement* ρ' *tel que, pour tout* $\mathsf{y} \in \mathcal{V}$ *tel que* $\mathsf{y} \in \mathcal{D}om(\rho)$ *:*

$$\rho'(\mathsf{y}) = \begin{cases} \mathsf{t} & si\ \mathsf{x} = \mathsf{y} \\ \rho(\mathsf{y}) & sinon \end{cases}$$

Nous notons $\mathcal{E}nv$ *l'ensemble de tous les environnements.*

Nous définissons un opérateur de surcharge afin d'éviter des conflits de variables lors de l'union de deux environnements.

Définition 13 (Surcharge). *Une opération de* surcharge $\rho_1 \lhd \rho_2$ *de deux environnements* ρ_1 *et* ρ_2 *est définie par :*

$$\rho_1 \lhd \rho_2(\mathsf{x}) = \begin{cases} \rho_1(\mathsf{x}) & si\ \mathsf{x} \notin \mathcal{D}om(\rho_2) \\ \rho_2(\mathsf{x}) & sinon \end{cases}$$

Lorsque nous manipulons plus d'un environnement, la notion de compatibilité sémantique est utile pour assurer qu'une même variable apparaissant dans ces environnements soit mappée à une même valeur.

Définition 14 (Compatibilité sémantique). *La* compatibilité sémantique *de deux environnements* ρ_1 *et* ρ_2, *dénotée par* $\rho_1 \bowtie \rho_2$ *est définie par :*

$$\forall \mathsf{x}\ (\mathsf{x} \in \mathcal{D}om(\rho_1) \wedge \mathsf{x} \in \mathcal{D}om(\rho_2)\ \Leftrightarrow \rho_1(\mathsf{x}) = \rho_2(\mathsf{x}))$$

Nous écrivons $\rho_1 \not\bowtie \rho_2$ pour signifier que ρ_1 et ρ_2 ne sont pas sémantiquement compatibles.

Un jugement d'évaluation pour les expressions Tom a la forme $\rho \Vdash \mathsf{t} \Downarrow (\rho', \dot{v})$ ou $\rho \Vdash \mathsf{t} \Downarrow (\{\rho'_1, \ldots, \rho'_n\}, \dot{v}')$ où $\{\rho'_1, \ldots, \rho'_n\}$ représente un ensemble d'environnements et $\dot{v}' \in \mathcal{V}al \backslash \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$. Les règles d'inférence pour la sémantique naturelle de Tom sont décrites dans le Figures 1, 2, 3 et 4. Elles sont considérées modulo renommage de variables. Nous admettons que les conditions numériques et de filtrage sur des termes construits à partir des opérateurs variadiques sont considérés modulo aplatissement. Les conditions de filtrage dépendent des jugements décrivant l'acceptation ou le rejet des motifs. Le filtrage de listes est une forme particulière de filtrage modulo $\mathcal{AU}$ où le motif et le sujet ont le même symbole

de tête. L'évaluation du filtrage de listes est décrite ici d'une « manière abstraite » tandis que sa description détaillée est donnée par le formalisme de la sémantique à petit pas défini par Gordon Plotkin [Plo81, Plo04]. Nous adoptons la sémantique à petit pas du filtrage modulo $\mathcal{AU}$ ainsi que celle de l'anti-filtrage pour le langage Tom décrite par Radu Kopetz dans le chapitre 3 de sa thèse [Kop08]. En outre, une discussion sur l'unification et le filtrage modulo $\mathcal{AU}$ parmi d'autres théories équationnelles peut être trouvée dans le travail de Jean-Pierre Jouannaud et Claude Kirchner [JK91]. Les conditions numériques dépendent des jugements décrivant l'acceptation ou le rejet des comparaisons entre les termes.

$$\frac{\mathsf{x} \in \mathcal{D}om(\rho) \quad \rho(\mathsf{x}) = \dot{u}}{\rho \Vdash \mathsf{x} \Downarrow (\rho, \dot{u})}\ \text{E-Var} \qquad \frac{}{\rho \Vdash \dot{v} \Downarrow (\rho, \dot{v})}\ \text{E-Val}$$

$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \dots \quad \rho \Vdash \mathsf{t}_n \Downarrow \dot{u}_n \quad \mathsf{f} \in \mathcal{F}_n}{\rho \Vdash \mathsf{f}(\mathsf{t}_1, \dots, \mathsf{t}_n) \Downarrow (\rho, \mathsf{f}(\dot{u}_1, \dots, \dot{u}_n))}\ \text{E-Fun}$$

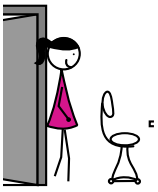
$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \dots \quad \rho \Vdash \mathsf{t}_n \Downarrow \dot{u}_n \quad \mathsf{v} \in \mathcal{F}^\star}{\rho \Vdash \mathsf{v}(\mathsf{t}_1, \dots, \mathsf{t}_n) \Downarrow (\rho, \flat(\ \mathsf{v}(\dot{u}_1, \dots, \dot{u}_n)))}\ \text{E-List}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad \rho \Vdash \mathsf{t} \Downarrow (\rho, \dot{u})}{\rho \Vdash (\mathsf{x} := \text{`}\mathsf{t}) \Downarrow (\rho \lhd \{\mathsf{x} \mapsto \dot{u}\}, \mathsf{accept})}\ \text{E-Ass} \qquad \frac{\rho \Vdash \mathsf{t} \Downarrow (\rho, \dot{u})}{\rho \Vdash \text{`}\mathsf{t} \Downarrow (\rho, \mathsf{accept})}\ \text{E-BqTerm}$$

où $\mathsf{t}, \mathsf{t}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ et $\dot{u}, \dot{u}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ pour $i \in [1, n]$.

FIGURE 1 – Règles d'évaluation pour des termes (avec backquote) et des affectations.

Dans la Figure 1, la règle [E-Var] décrit l'évaluation d'une variable liée à une valeur dans l'environnement : la variable se réduit à cette valeur sans modifier l'environnement. Pour l'évaluation d'une valeur, la règle [E-Val] retourne l'environnement et la valeur d'origine. Les règles [E-Fun] et [E-List] correspondent respectivement au dépliage des opérateurs syntaxiques et variadiques. En outre, la règle [E-List] considère les listes aplaties, *i*.e. les listes qui n'ont pas de sous-listes imbriquées avec un même symbole de tête. Le résultat est obtenu par l'évaluation de chaque argument de la fonction. Pour l'évaluation d'une instruction d'affectation, la règle [E-Ass] incrémente l'environnement à travers l'assignation de la valeur de la variable au résultat de l'évaluation du terme avec backquote dans l'argument de gauche de l'opérateur :=. La règle [E-BqTerm] décrit l'évaluation des termes Tom apparaissant dans du code Java. En tant que sémantique approximative de termes Java combinés avec des termes Tom, nous considérons que les variables ou les fonctions Java représentées à l'origine par des termes Tom ne s'évaluent qu'à des termes clos de Tom.



$$\frac{\mathsf{x} \in \mathcal{V} \quad \tau =_t \mathsf{typeof}([\,],\mathsf{x}) \quad \mathsf{typeof}([\,],\dot{u}) <:_t \tau}{\rho \Vdash \mathsf{x} \ll |\tau|\ \dot{u} \Downarrow (\{\rho \lhd \{\mathsf{x} \mapsto \dot{u}\}\}, \mathsf{accept})}\ \text{E-MSubVarAcc}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad \tau \neq_t \mathsf{typeof}([\,],\mathsf{x}) \vee \neg(\mathsf{typeof}([\,],\dot{u}) <:_t \tau)}{\rho \Vdash \mathsf{x} \ll |\tau|\ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MSubVarRej}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad symb(\dot{u}) \in \mathcal{F}^\star \quad \tau =_t \mathsf{typeof}([\,],\mathsf{x}) \quad \tau =_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x}^* \ll |\tau|\ \dot{u} \Downarrow (\{\rho \lhd \{\mathsf{x} \mapsto \dot{u}\}\}, \mathsf{accept})}\ \text{E-MSVarAcc}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad symb(\dot{u}) \in \mathcal{F}^\star \quad \tau \neq_t \mathsf{typeof}([\,],\mathsf{x}) \vee \tau \neq_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x}^* \ll |\tau|\ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MSVarRej}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad \rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\rho', \dot{v})}{\rho \Vdash \mathsf{x@t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho' \lhd \{\mathsf{x} \mapsto \dot{u}\}\}, \dot{v})}\ \text{E-MAlias}$$

$$\frac{\begin{array}{c}\rho_0 \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \dot{u}_1) \Downarrow (\rho_1,\mathsf{accept}) \quad \ldots \quad \rho_0 \Vdash (\mathsf{t}_n \ll |\tau_n|\ \dot{u}_n) \Downarrow (\rho_n,\mathsf{accept}) \\ \mathsf{f} \in \mathcal{F} \quad \tau =_t \mathsf{typeof}([\,],\mathsf{f}(\dot{u}_1,\ldots,\dot{u}_n)) \quad \forall i \in [1,n], \rho_{i-1} \bowtie \rho_i\end{array}}{\rho_0 \Vdash \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n) \ll |\tau|\ \mathsf{f}(\dot{u}_1,\ldots,\dot{u}_n) \Downarrow (\{\bigcup_{j=0}^{n} \rho_j\}, \mathsf{accept})}\ \text{E-MFunAcc}$$

$$\frac{\begin{array}{c}\rho_0 \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \dot{u}_1) \Downarrow (\rho_1,\dot{v}_1) \quad \ldots \quad \rho_0 \Vdash (\mathsf{t}_n \ll |\tau_n|\ \dot{u}_n) \Downarrow (\rho_n,\dot{v}_n) \\ \mathsf{f} \in \mathcal{F} \quad \exists i \in [1,n], (\dot{v}_i = \mathsf{reject} \vee \rho_{i-1} \not\bowtie \rho_i)\end{array}}{\rho_0 \Vdash \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n) \ll |\tau|\ \mathsf{f}(\dot{u}_1,\ldots,\dot{u}_n) \Downarrow (\{\rho_0\}, \mathsf{reject})}\ \text{E-MFunRej}$$

$$\frac{\mathsf{g},\ \mathsf{g'} \in \mathcal{F} \cup \mathcal{F}^\star \quad \mathsf{g} \neq \mathsf{g'} \vee \tau \neq_t \mathsf{typeof}([\,],\mathsf{g'}(\dot{u}_1,\ldots,\dot{u}_n))}{\rho \Vdash \mathsf{g}(\mathsf{t}_1,\ldots,\mathsf{t}_n) \ll |\tau|\ \mathsf{g'}(\dot{u}_1,\ldots,\dot{u}_n) \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MFunListRej}$$

$$\frac{\rho \Vdash \mathsf{t'} \Downarrow (\rho, \dot{u}) \quad \rho \Vdash (\mathsf{t} \ll |\tau|\ \dot{u})) \Downarrow (\rho', \dot{v}) \quad \mathcal{V}ar(\mathsf{t'}) \neq \emptyset}{\rho \Vdash (\mathsf{t} \ll |\tau|\ \mathsf{t'}) \Downarrow (\{\rho'\}, \dot{v})}\ \text{E-MSubj}$$

où $\mathsf{t}, \mathsf{t}_i, \mathsf{t'}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ et $\dot{u}, \dot{u}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ pour $i \in [1,n]$.

FIGURE 2 – Règles d'évaluation pour de conditions de filtrage syntaxique.

La Figure 2 présente les règles dont les noms sont préfixés par « E-M ». Elles représentent la sémantique du filtrage de motifs et produisent un ensemble non vide d'environnements. Les types τ apparaissant dans des conditions de filtrage correspondent à l'annotation de type explicite dont le motif devrait être. Le type d'un terme t est obtenu par un jugement auxiliaire typeof([ ],t) qui sera discuté en détail dans la section suivante et où [ ] représente une séquence vide. Bien que l'introduction d'un ordre partiel sur les types ne modifie pas les expressions Tom elles-mêmes, cela a un impact sur l'évaluation des conditions de filtrage puisque la construction **%match** est paramétrée par un déclaration facultative de sorte $|\tau|$. La description de l'évaluation du filtrage de motifs en considérant le sous-typage a été inspiré par le travail de Delia Kesner [Kes91]. Plus que de la vérification du filtrage structurel, la relation de réduction $\Downarrow$ sur les expressions doit vérifier que les types des instances de motifs sont de sous-types des types attendus pour ces motifs.

$$\frac{\rho \Vdash \mathsf{t}[\mathsf{x}]_\omega \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \mathsf{accept}) \quad \rho' \Vdash \mathit{not}(\mathsf{t}[\mathsf{t}_1]_\omega \ll |\tau|\ \dot{u}) \Downarrow (\{\rho'\}, \mathsf{accept}) \quad \mathsf{x} \in \mathcal{V}\backslash\mathcal{V}ar(\mathsf{t}) \quad \mathsf{t}_{|\omega} = !\mathsf{t}_1}{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \mathsf{accept})}\ \text{E-MAntiAcc}$$

$$\frac{\rho \Vdash \mathsf{t}[\mathsf{x}]_\omega \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \dot{v}_1) \quad \rho' \Vdash \mathit{not}(\mathsf{t}[\mathsf{t}_1]_\omega \ll |\tau|\ \dot{u}) \Downarrow (\{\rho'\}, \dot{v}_2) \quad \mathsf{x} \in \mathcal{V}\backslash\mathcal{V}ar(\mathsf{t}) \quad \mathsf{t}_{|\omega} = !\mathsf{t}_1 \quad \exists i \in [1,2], \dot{v}_i = \mathsf{reject}}{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MAntiRej}$$

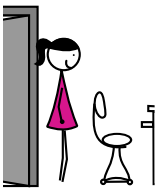
$$\frac{\rho \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \mathsf{t'}_1) \Downarrow (\rho_1, \mathsf{accept}) \quad \rho_1 \Vdash (\mathsf{t}_2 \ll |\tau_2|\ \mathsf{t'}_2) \Downarrow (\rho_2, \mathsf{accept}) \quad \rho \bowtie \rho_1 \quad \rho_1 \bowtie \rho_2 \quad \mathit{symb}(\mathsf{t}) \in \mathcal{F}^\star \vee \mathit{symb}(\mathsf{t'}) \in \mathcal{F}^\star}{\rho \Vdash \mathsf{t} \ll |\tau|\ \mathsf{t'} \Downarrow (\{\rho_2\}, \mathsf{accept})}\ \text{E-MListAcc}$$

$$\frac{\rho \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \mathsf{t'}_1) \Downarrow (\rho_1, \dot{v}_1) \quad \rho_1 \Vdash (\mathsf{t}_2 \ll |\tau_2|\ \mathsf{t'}_2) \Downarrow (\rho_2, \dot{v}_2) \quad \exists i \in [1,2], \dot{v}_i = \mathsf{reject} \quad \mathit{symb}(\mathsf{t}) \in \mathcal{F}^\star \vee \mathit{symb}(\mathsf{t'}) \in \mathcal{F}^\star}{\rho \Vdash \mathsf{t} \ll |\tau|\ \mathsf{t'} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MListRej}$$

where $\mathsf{t}, \mathsf{t}_i, \mathsf{t'}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ and $\dot{u}, \dot{u}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ for $i \in [1, n]$.

FIGURE 3 – Règles d'évaluation pour de conditions de filtrage (d'anti-motifs) associatif.

Les règles du filtrage associatif et du filtrage d'anti-motifs sont présentées dans la Figure 3. Elles ne sont pas dirigée par la syntaxe puisque les termes apparaissant dans leurs prémisses ne sont pas explicitement liés à ceux de leurs conclusions. Les règles [E-MListAcc] et [E-MListRej] décrivent l'évaluation du filtrage de listes. Leurs prémisses dépendent de l'application de l'une des règles sémantiques à petit pas présentées dans le chapitre 3 de la thèse de Radu Kopetz [Kop08] : Mutate, $\mathsf{SymbolClash}_1^+$ ou $\mathsf{SymbolClash}_2^+$. Chacune de ces règles a



deux prémisses contenant des termes t_i et $\mathsf{t'}_i$ composées non seulement des termes t et $\mathsf{t'}$ dans la conclusion, mais aussi de termes frais. De même, les règles [E-MAntiAcc] et [E-MAntiRej] pour l'évaluation d'une condition d'anti-filtrage (*i*.e. une condition de filtrage contenant des anti-motifs) correspondent à la règle sémantique à petit pas ElimAnti définie par Radu Kopetz.

Le sous-terme *not* apparaissant dans leurs prémisses n'appartient pas à la syntaxe abstraite de base de Tom (c.f. Définition 9) et son évaluation est effectuée par l'algorithme $\mathcal{AU}$-AntiMatch défini dans la thèse de Radu Kopetz. Ici, nous simulons l'évaluation d'un terme ayant *not* comme symbole de tête par deux règles auxiliaires d'évaluation [E-NotAcc] and [E-NotRej] :

$$\frac{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \dot{v}) \quad (\dot{v} = \mathsf{reject} \vee \rho \not\bowtie \rho')}{\rho \Vdash \mathit{not}(\mathsf{t} \ll |\tau|\ \dot{u}) \Downarrow (\{\rho\}, \mathsf{accept})}\ \text{E-NotAcc}$$

et

$$\frac{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \dot{v}) \quad \dot{v} = \mathsf{accept} \quad \rho \bowtie \rho'}{\rho \Vdash \mathit{not}(\mathsf{t} \ll |\tau|\ \dot{u}) \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-NotRej}$$

L'algorithme $\mathcal{AU}$-AntiMatch donne également une priorité plus élevée aux règles [E-MAntiAcc] et [E-MAntiRej]. Ainsi, une condition d'anti-filtrage $\mathsf{f}(\,!\mathsf{x}) \ll |\tau|\ \dot{u}$ doit d'abord être évaluée soit par [E-MAntiAcc], soit par [E-MAntiRej]. Les autres règles, telles que [E-MFunAcc] et [E-MFunRej] sont donc appliquées ensuite.

Les règles de sémantique décrivant l'évaluation des expressions qui ne sont ni des termes ni les conditions de filtrage sont présentées dans la Figure 4. Pour les conditions numériques, les règles [E-NumAcc] et [E-NumRej] s'appliquent pour évaluer les deux termes comparés par l'opérateur $\diamond$ et produisent un singleton contenant l'environnement inchangé ρ. Les conditions composées de conjonctions et/ou disjonctions (imbriquées) sont considérées en forme normale disjonctive. Chaque condition d'une conjonction est évaluée soit par la règle [E-ConjAcc], soit par la règle [E-ConjRej]. L'évaluation d'une séquence d'actions apparaissant dans le côté droit d'une règle réduit toujours à accept par la règle [E-Action]. Le résultat de son application ne dépend pas de l'évaluation des actions car celles-ci sont toutes possiblement mais pas nécessairement exécutées. L'évaluation d'une disjonction des conditions dans le côté gauche de règles est décrite par les règles [E-ExDisjAcc], [E-DisjAcc] et [E-DisjRej] qui peuvent produire un nouvel ensemble d'environnements. Cela arrive puisque l'évaluation d'une règle $\mathit{cond}_1 \vee \mathit{cond}_2 \longrightarrow \mathit{action}$ est sémantiquement équivalente à l'évaluation d'un bloc contenant deux règles Tom $\{\mathit{cond}_1 \longrightarrow \mathit{action}\,;\, \mathit{cond}_2 \longrightarrow \mathit{action}\}$ dans deux environnements différents. Les blocs sont traités par la règle [E-Rule] qui décrit l'évaluation d'une règle (Tom) en fonction de l'évaluation essentiellement du côté gauche et éventuellement du côté droit. La sémantique d'un bloc de règles Tom est différente de la plupart des langages fonctionnels : quand le côté

$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \rho \Vdash \mathsf{t}_2 \Downarrow \dot{u}_2 \quad \dot{u}_1 \diamond \dot{u}_2}{\rho \Vdash (\mathsf{t}_1 \diamond\, \mathsf{t}_2) \Downarrow (\{\rho\}, \mathsf{accept})}\ \text{E-NumAcc}$$

$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \rho \Vdash \mathsf{t}_2 \Downarrow \dot{u}_2 \quad \neg(\dot{u}_1 \diamond \dot{u}_2)}{\rho \Vdash (\mathsf{t}_1 \diamond\, \mathsf{t}_2) \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-NumRej}$$

$$\frac{\rho \Vdash \mathit{cond}_1 \Downarrow (\{\rho_1\}, \mathsf{accept}) \quad \rho_1 \Vdash \mathit{cond}_2 \Downarrow (\{\rho_2\}, \mathsf{accept})}{\rho \Vdash \mathit{cond}_1 \wedge \mathit{cond}_2 \Downarrow (\{\rho_2\}, \mathsf{accept})}\ \text{E-ConjAcc}$$

$$\frac{\begin{array}{c}\rho \Vdash \mathit{cond}_1 \Downarrow (\{\rho_1\}, \dot{v}_1) \quad \rho_1 \Vdash \mathit{cond}_2 \Downarrow (\{\rho_2\}, \dot{v}_2) \\ \exists i \in [1,2], \dot{v}_i = \mathsf{reject}\end{array}}{\rho \Vdash \mathit{cond}_1 \wedge \mathit{cond}_2 \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-ConjRej}$$

$$\frac{\begin{array}{c}\rho \Vdash \mathit{cond}_1 \Downarrow (\{\rho_{1,1}, \ldots, \rho_{1,n_1}\}, \dot{v}_1) \quad \rho \Vdash \mathit{cond}_2 \Downarrow (\{\rho_{2,1}, \ldots, \rho_{2,n_2}\}, \dot{v}_2) \\ \exists i, j \in [1,2], (\dot{v}_i = \mathsf{accept} \wedge \dot{v}_j = \mathsf{reject} \wedge i \neq j)\end{array}}{\rho \Vdash \mathit{cond}_1 \vee \mathit{cond}_2 \Downarrow (\{\rho_{i,1}, \ldots, \rho_{i,n_i}\}, \mathsf{accept})}\ \text{E-ExDisjAcc}$$

$$\frac{\rho \Vdash \mathit{cond}_1 \Downarrow (\{\rho_1, \ldots, \rho_n\}, \mathsf{accept}) \quad \rho \Vdash \mathit{cond}_2 \Downarrow (\{\rho'_1, \ldots, \rho'_m\}, \mathsf{accept})}{\rho \Vdash \mathit{cond}_1 \vee \mathit{cond}_2 \Downarrow (\{\rho_1, \ldots, \rho_n\} \cup \{\rho'_1, \ldots, \rho'_m\}, \mathsf{accept})}\ \text{E-DisjAcc}$$

$$\frac{\rho \Vdash \mathit{cond}_1 \Downarrow (\{\rho_1, \ldots, \rho_n\}, \mathsf{reject}) \quad \rho \Vdash \mathit{cond}_2 \Downarrow (\{\rho'_1, \ldots, \rho'_m\}, \mathsf{reject})}{\rho \Vdash \mathit{cond}_1 \vee \mathit{cond}_2 \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-DisjRej}$$

$$\frac{\rho \Vdash \mathit{action}_1 \Downarrow (\rho_1, \dot{v}_1) \quad \ldots \quad \rho_{n-1} \Vdash \mathit{action}_n \Downarrow (\rho_n, \dot{v}_n)}{\rho \Vdash (\mathit{action}_1; \ldots; \mathit{action}_n) \Downarrow (\rho_n, \mathsf{accept})}\ \text{E-Action}$$

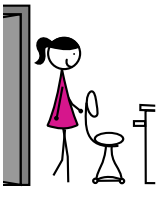
$$\frac{\begin{array}{c}\rho \Vdash \mathit{cond} \Downarrow (\{\rho_1, \ldots, \rho_n\}, \dot{v}) \\ \rho_1 \Vdash \mathit{action} \Downarrow (\rho'_1, \mathsf{accept}) \quad \ldots \quad \rho_n \Vdash \mathit{action} \Downarrow (\rho'_n, \mathsf{accept})\end{array}}{\rho \Vdash \mathit{cond} \longrightarrow \mathit{action} \Downarrow (\rho, \dot{v})}\ \text{E-Rule}$$

$$\frac{\rho \Vdash \mathit{rule}_1 \Downarrow (\rho, \dot{v}_1) \quad \ldots \quad \rho \Vdash \mathit{rule}_n \Downarrow (\rho, \dot{v}_n) \quad \exists i, \dot{v}_i = \mathsf{accept}}{\rho \Vdash \{\mathit{rule}_1; \ldots; \mathit{rule}_n\} \Downarrow (\rho, \mathsf{accept})}\ \text{E-BlockAcc}$$

$$\frac{\rho \Vdash \mathit{rule}_1 \Downarrow (\rho, \dot{v}_1) \quad \ldots \quad \rho \Vdash \mathit{rule}_n \Downarrow (\rho, \dot{v}_n) \quad \forall i, \dot{v}_i = \mathsf{reject}}{\rho \Vdash \{\mathit{rule}_1; \ldots; \mathit{rule}_n\} \Downarrow (\rho, \mathsf{reject})}\ \text{E-BlockRej}$$

où $\mathsf{t}_1, \mathsf{t}_2 \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ et $\dot{u}_1, \dot{u}_2 \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$.

FIGURE 4 – Règles d'évaluation pour de conditions numériques et d'expressions résultantes.



gauche d'une règle est accepté, l'action dans la partie droite est évaluée et la règle suivante est ensuite évaluée, à moins que le flot d'exécution soit interrompu. Ceci est décrit respectivement par les règles [E-BlockAcc] et [E-BlockRej] qui résultent en accept si au moins une règle Tom est acceptée et en reject sinon.

Vérification de types

Les constructeurs de types et de données correspondent respectivement à des sortes de base et des opérateurs (syntaxique et variadique). Les programmes Tom sont considérés comme des programmes avec l'annotation de type explicite. Cela permet la décidabilité de l'évaluation des informations de type des expressions Tom durant la compilation par l'utilisation du typage statique. Or, l'indication « un terme t est de type τ » signifie que nous pouvons statiquement conclure que t s'évalue à une valeur de type τ. Afin d'éliminer les programmes Tom qui causent des erreurs lors de l'exécution, nous décrivons un système de vérification de types qui vérifie automatiquement que ces programmes sont bien-typés. En outre, l'ajout de sous-typage permet au typage statique de rejeter moins de programmes qui ont un bon comportement par l'affaiblissement de la notion de compatibilité de types. Dans un système de types nominal comme le nôtre, le contexte peut être enrichi avec des déclarations de sous-typage permettant aux termes d'être typés avec des sous-types des types indiqués dans leurs annotations de type.

Définitions préliminaires

Les règles de typage de notre système de vérification de types manipulent un *contexte* défini comme un ensemble de paires (variable,type), (opérateur,rang) et (type,type), ceux derniers représentant des relations de sous-typage.

Définition 15 (Contexte avec sous-types)**.** *Un* contexte *est défini par :*

$$\Gamma ::= \varnothing \mid \Gamma; s_1^{h_1} <:_t s_2^{h_2} \mid \Gamma; \mathsf{x} : \tau \mid \Gamma; \mathsf{f} : s_1^?, \ldots, s_n^? \to s^? \mid \Gamma; \mathsf{v} : (s_1^?)^\star \to s^v$$

où $s_1^{h_1}, s_2^{h_2} \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\})$.

Un contexte Γ associe des types à des variables. Il associe également des rangs aux opérateurs syntaxiques et variadiques. Nous désignons par $\Gamma(\gamma)$ le fait qu'une paire γ appartient à Γ. Le contexte a au plus une déclaration de type pour chaque variable et un rang par opérateur puisque la surcharge d'opérateur est interdite. L'application de la clôture transitive de $<:_t$ à toutes les déclarations de sous-typage trouvées dans Γ génère Γ^*. L'initialisation du contexte se produit pendant l'analyse syntaxique : chaque variable d'une condition (de filtrage ou numérique) est associée à sa sorte décorée avec ? tandis que chaque opérateur algébrique d'un ancrage Tom est associé à son rang. Les rangs des opérateurs algébriques sont construits à partir de la décoration de leur sorte de la manière suivante :

- pour un opérateur syntaxique f : toutes les sortes qui apparaissent dans le rang sont décorées avec ?,
- pour un opérateur variadique v : toutes les sortes qui apparaissent dans le domaine sont décorées avec ? alors que la sorte du codomaine est décorée avec v.

En fait, Tom ignore le code Java, mais dans le but de vérifier les types, nous considérons que le rang des opérateurs Java et les types des variables Java sont connus. Par conséquent, les opérateurs Java (représentés par des opérateurs Tom) et leurs rangs ainsi que les variables Java et leurs types sont également déclarés dans le contexte. En outre, l'accès au contexte est défini par la fonction typeof(Γ,t) qui retourne le type associé au terme t dans le contexte Γ.

Définition 16 (Accès au contexte)**.** L'accès au contexte *est fait par une fonction binaire partielle* $\mathsf{typeof} : \Gamma \times \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V}) \to \mathcal{T}y$ *définie par :*

$$\mathsf{typeof}(\Gamma,\mathsf{x}) = \tau,\ si\ \mathsf{x} : \tau \in \Gamma \qquad \mathsf{typeof}(\Gamma,\mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) = s^?,\ si\ \mathsf{f} : s_1^?,\ldots,s_n^? \to s^? \in \Gamma$$
$$\mathsf{typeof}(\Gamma,\mathsf{x@t}) = \mathsf{typeof}(\Gamma,\mathsf{t}) \qquad \mathsf{typeof}(\Gamma,\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) = s^v,\ si\ \mathsf{v} : (s_1^?)^\star \to s^v \in \Gamma$$
$$\mathsf{typeof}(\Gamma,!\mathsf{t}) = \mathsf{typeof}(\Gamma,\mathsf{t})$$

où $\mathsf{x} \in \mathcal{V},\ \mathsf{f} \in \mathcal{F},\ \mathsf{v} \in \mathcal{F}^\star,\ s^?, s_i^?, s^v \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\})$ *et* $\mathsf{t},\mathsf{t}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *pour* $i \in [1,n]$.

Règles de vérification de types

Un jugement de typage pour du code Tom a la forme $\Gamma \vdash e : \tau$ et est défini par un ensemble de règles d'inférence qui associent des types à des expressions Tom. Ces règles sont présentées dans la Figure 5. Les règles de typage sont standard, sauf pour l'utilisation de sortes décorées comme étant des types. Toutes les variables d'une règle de typage doivent être considérées comme quantifiées universellement. En outre, nous considérons les variables apparaissant dans les motifs de différentes expressions *rule* modulo renommage. À partir d'un contexte Γ et d'une expression Tom e, on dit que e est *bien-typé* s'il est possible de construire une dérivation de typage pour lui.

Définition 17 (Expression et code bien-typés)**.** *Une expression Tom* e *est* bien-typée *dans un contexte* Γ *s'il existe un type* $\tau \in \mathcal{T}y \backslash \mathcal{X}$ *tel que* $\Gamma \vdash e : \tau$.

Un code Tom est bien-typé *dans un contexte* Γ *si et seulement si toutes ses expressions sont bien-typées.*

Dans la Figure 5, les relations de sous-typage sur de types sont traitées dans les jugements de typage $\Gamma \vdash e : \tau$ par une règle de subsomption [Sub] qui indique que chaque terme t de type s_1^h est également de type $s^?$ si $s_1^h <:_t s^?$. Les règles le plus particulièrement intéressantes sont celles qui s'appliquent aux listes. Elles sont au nombre de trois : [T-Empty] vérifie si une liste vide est du même type qui celui qui a été déclaré dans Γ ; [T-Elem] est similaire à [T-Fun], mais est appliquée aux opérateurs variadiques ; et [T-Merge] est appliquée à une concaténation de deux listes de types s^v dans Γ, ce qui résulte en une nouvelle liste de même type s^v.

$$\frac{}{\Gamma(\mathsf{x} : s^h) \vdash \mathsf{x} : s^h}\ \text{T-Var} \qquad \frac{}{\Gamma(\mathsf{x} : s^v) \vdash \mathsf{x*} : s^v}\ \text{T-SVar}$$

$$\frac{\Gamma \vdash e : s_1^h}{\Gamma(s_1^h <:_t s^?) \vdash e : s^?}\ \textbf{Sub} \qquad \frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma(\mathsf{x} : s^h) \vdash \mathsf{x@t} : s^h}\ \text{T-Alias}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : s_1^? \quad \ldots \quad \Gamma \vdash \mathsf{t}_n : s_n^?}{\Gamma(\mathsf{f} : s_1^?, \ldots, s_n^? \to s^?) \vdash \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^?}\ \text{T-Fun} \qquad \frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma \vdash \mathsf{!t} : s^h}\ \text{T-Anti}$$

$$\frac{}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \mathsf{v}() : s^v}\ \text{T-Empty}$$

$$\frac{\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v \qquad \Gamma \vdash \mathsf{t}_n : s_1^?}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v}\ \text{T-Elem}$$
$$\text{si } \mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v \text{ et } \mathsf{t}_n \neq \mathsf{x*}$$

$$\frac{\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v \qquad \Gamma \vdash \mathsf{t}_n : s^v}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v}\ \text{T-Merge}$$
$$\text{si } \mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v \text{ ou } \mathsf{t}_n = \mathsf{x*}$$

$$\frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma(\mathsf{x} : s^h) \vdash \mathsf{x} := \text{`}\mathsf{t} : wt}\ \text{T-Ass} \qquad \frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma \vdash \text{`}\mathsf{t} : wt}\ \text{T-BqTerm}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : s^? \qquad \Gamma \vdash \mathsf{t}_2 : s^?}{\Gamma \vdash (\mathsf{t}_1 \ll |s^h|\ \mathsf{t}_2) : wt}\ \text{T-Match} \qquad \frac{\Gamma \vdash \mathsf{t}_1 : s^? \qquad \Gamma \vdash \mathsf{t}_2 : s^?}{\Gamma \vdash (\mathsf{t}_1 \diamond\ \mathsf{t}_2) : wt}\ \text{T-Num}$$
$$\text{si } \mathsf{typeof}(\Gamma,\mathsf{t}_i) = s^{h_i} \text{ for } i \in [1,2]$$

$$\frac{\Gamma \vdash (cond_1) : wt \qquad \Gamma \vdash (cond_2) : wt}{\Gamma \vdash (cond_1 \wedge cond_2) : wt}\ \text{T-Conj}$$

$$\frac{\Gamma \vdash (cond_1) : wt \qquad \Gamma \vdash (cond_2) : wt}{\Gamma \vdash (cond_1 \vee cond_2) : wt}\ \text{T-Disj}$$

$$\frac{\Gamma \vdash action_1 : wt \quad \ldots \quad \Gamma \vdash action_n : wt}{\Gamma \vdash (action_1;\ldots;action_n) : wt}\ \text{T-Action}$$

$$\frac{\Gamma \vdash cond : wt \qquad \Gamma \vdash action : wt}{\Gamma \vdash (cond \longrightarrow action) : wt}\ \text{T-Rule}$$

$$\frac{\Gamma \vdash rule_1 : wt \quad \ldots \quad \Gamma \vdash rule_n : wt}{\Gamma \vdash \{rule_1;\ldots;rule_n\} : wt}\ \text{T-Block}$$

où $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$, $\mathsf{v} \in \mathcal{F}^\star$ et $h \in \mathcal{F}^\star \cup \{?\}$

FIGURE 5 – Règles de vérification de types pour les expressions Tom.

Les règles de vérification de types construisent seulement des expressions sans vérifier les types de termes, à l'exception des règles [T-Match] et [T-Num] qui introduisent une condition (de filtrage ou numérique) bien-typée si les deux termes qui y apparaissent sont d'un même type $s^?$. La majorité des règles de vérification de types est dirigée par la syntaxe sauf celles ayant une condition d'application qui utilise la fonction typeof. L'algorithme de vérification de types lit les dérivations de bas en haut. Puisque la règle [Sub] peut être appliquée à tout genre de termes, nous considérons une stratégie qui l'applique si et seulement si aucune autre règle de typage ne peut être appliquée. Dans la pratique, [Sub] est combinée avec [T-Anti], [T-Alias], [T-Fun], [T-Elem] et [T-Merge] et le type s^h figurant dans sa prémisse est défini en fonction du résultat de la fonction typeof(Γ,e). Les règles de typage sont appliquées exhaustivement jusqu'à ce que chaque branche de dérivation atteigne des règles sans prémisse, ou jusqu'à ce qu'aucun progrès ne puisse être fait. Le premier cas assure que l'expression originale est bien-typée tandis que le dernier cas caractérise une erreur de type.

Inférence de types

L'algorithme de vérification de types présenté dans la section précédente dépend des annotations de type explicites, même pour les variables. Mais, dans Tom, seul les rangs des opérateurs sont connus. En outre, le système de vérification de types présenté précédemment nécessite des règles pour contrôler son utilisation afin de trouver l'arbre de déduction prévu pour une expression. Sans ces règles, il est possible de trouver plus d'un arbre de déduction pour la même expression par l'application de [Sub]. Pour cette raison, nous nous intéressons à accroître notre système de types en définissant un autre algorithme capable d'inférer statiquement les types des termes Tom. Par conséquent, nous développons un système d'inférence de types plus puissant, capable de calculer un *type principal* pour un code Tom dont certaines des annotations de type n'ont pas été spécifiées.

Définitions préliminaires

Comme on le voit dans la section précédente, l'ensemble de types $\mathcal{T}y$ utilisés par notre système de types inclut des variables de type α en tant que types non interprétés. En fait, les variables de type sont des éléments de remplissage pour des types clos dont les identités exactes ne sont pas pertinentes dans les jugements de typage et peuvent être instanciées avec d'autres types. Nous définissons une *substitution de type* σ des variables de type par des types comme le fait Benjamin Pierce dans son travail [Pie02].

Définition 18 (Substitution de type)**.** *Une* substitution de type *(ou simplement* substitution*, quand il est clair que nous parlons de types) est une fonction totale de $\mathcal{X}$ dans $\mathcal{T}y$, i.e. des variables de type dans les types.*

Nous notons $\mathcal{D}om(\sigma)$ l'ensemble des variables de type apparaissant dans les côtés gauches des paires de σ, et $\mathcal{R}ange(\sigma)$ l'ensemble des types apparaissant dans le côtés droits. Si $\mathcal{D}om(\sigma) = \{\alpha_1, \ldots, \alpha_n\}$, alors nous notons σ par $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$. Nous notons que la même variable peut apparaitre à la fois dans le domaine et dans l'image d'une substitution.

L'application d'une substitution de type à un contexte est définie par :

$$\sigma(\ldots, \mathsf{x}_1 : \alpha_1, \ldots, \mathsf{x}_n : \alpha_n, \ldots) = (\ldots, \mathsf{x}_1 : \sigma\alpha_1, \ldots, \mathsf{x}_n : \sigma\alpha_n, \ldots)$$

où seules les variables peuvent être typées avec des variables de type dans le contexte, puisque le rang de tous les operateurs (algébriques et concrets) constituent des annotations de type. Des substitutions de type préservent la validité des jugements de typage dont l'expression Tom contient des variables.

La fonction de nettoyage de décoration introduite dans la Définition 7 peut être appliquée sur des variables de type en tenant compte des substitutions de type :

$$|\alpha| = |s^h|, \text{ s'il existe une substitution } \sigma \text{ tel que } \sigma\alpha = s^h$$

où $\alpha \in \mathcal{X}$.

Pour déterminer si un code Tom contenant des variables de type est bien-typé, il faut décrire la façon dont les variables de type peuvent être instanciées par des types, à la fois dans l'expression et dans le contexte. Il faut aussi construire une dérivation de typage valide pour l'expression.

Définition 19 (Solution pour un jugement de typage)**.** *Soit Γ un contexte et e une expression Tom. Une* solution *pour (Γ, e) est une paire (σ, τ) telle que $\sigma\Gamma \vdash \sigma e : \tau$, où $\tau \in \mathcal{T}y \backslash \mathcal{X}$.*

Plus qu'une simple vérification de solutions pour (Γ, e) à travers la construction de dérivations de typage, nous nous intéressons à trouver ces solutions en contraignant les variables de type qui apparaissent dans les dérivations de typage. Nous visons à restreindre l'ensemble des valeurs possibles des variables de type à travers le calcul d'un ensemble de *contraintes*. Ainsi, ces contraintes limitent les types et, en conséquence, les sortes de base (*i*.e. les types de données algébriques) que les termes peuvent avoir.

Définition 20 (Ensemble de contraintes)**.** *Considérons l'ensemble de types $\mathcal{T}y$.* L'ensemble de contraintes $\mathcal{C}$ *est un ensemble contenant de contraintes définies par la grammaire algébrique suivante :*

$$c ::= \tau_1 =_t \tau_2 \mid \tau_1 <:_t \tau_2$$

où $c \in \mathcal{C}$, $\tau_1, \tau_2 \in \mathcal{T}y$.

Une contrainte close *est une contrainte sans variables, i.e. une contrainte construite à partir des types clos.*

On dit qu'une substitution σ satisfait *une équation $\tau_1 =_t \tau_2$ si $\sigma(\tau_1) =_t \sigma(\tau_2)$. De même, on dit que σ* satisfait *une contrainte de sous-typage $\tau_1 <:_t \tau_2$ si $\sigma(\tau_1) <:_t \sigma(\tau_2)$. Ainsi, σ satisfait $\mathcal{C}$ s'il satisfait toutes les contraintes de $\mathcal{C}$. Ceci est noté par $\sigma \models \mathcal{C}$.*

L'ensemble $\mathcal{V}ar(\mathcal{C})$ dénote l'ensemble de variables de type figurant dans $\mathcal{C}$.

Règles d'inférence de types

Un jugement de typage basé sur des contraintes pour les expressions Tom a la forme $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ et est défini par un ensemble de règles d'inférence associant des types à des expressions Tom. Ces règles sont présentées dans les Figures 6 et 7. Des contraintes d'égalité et de sous-typage sont calculées en fonction de l'application de ces règles. $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ peut être lu, de manière informelle, comme « l'expression e est de type τ sous des hypothèses Γ lorsque les contraintes $\mathcal{C}$ sont satisfaites ». Formellement, les jugements de typage attestent que

$$\forall \sigma (\sigma \models \mathcal{C} \Rightarrow \sigma\Gamma \vdash \sigma e : \sigma\tau)$$

Afin d'inférer le type d'une expression donnée e, le contexte Γ est initialisée avec :

1. les relations de sous-typage de la forme $s_1^? <:_t s_2^?$ où $s_1^? <:_t s_2^?$
2. une paire de la forme $(\mathsf{f} : s_1^?, \ldots, s_n^? \to s^?)$ pour chaque opérateur syntaxique f apparaissant dans le jugement de typage où $s_i^?, s^? \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\})$ pour $i \in [1, n]$,
3. une paire de la forme $(\mathsf{v} : s_1^{?\star} \to s^v)$ pour chaque opérateur variadique v apparaissant dans le jugement de typage où $s_1^?, s^v \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\})$,
4. une paire de la forme $(\mathsf{x} : \alpha)$ pour chaque variable x et variable étoile x* apparaissant dans le jugement de typage où $\alpha \in \mathcal{X}$ est une variable de type fraiche.

Chaque variable de type introduite dans une sous-dérivation est une variable de type fraiche et les variables de type fraiches sont distinctes dans des différentes sous-dérivations. Comme dans le système de vérification de types, nous expliquons les règles de la Figure 6 qui concernent les listes. Dans la règle [CT-Empty], si v est un symbole de fonction variadique avec codomaine s^v, alors un type code α qui est égal à s^v est un type pour une liste vide v(). L'insertion d'un élément dans une liste est traitée par la règle [CT-Elem] : pour une liste

$$\frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Var} \qquad \frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x*} : \alpha) \vdash_{ct} \mathsf{x*} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-SVar}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash_{ct} \mathsf{!t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Anti} \qquad \frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \quad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x@t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Alias}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \quad \dots \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_n \bullet \mathcal{C}_n \qquad \mathcal{C} = \{\alpha_0 =_t s^?\} \bigcup_{i=1}^{n} \mathcal{C}_i \cup \{\alpha_i <:_t s_i^?\}}{\Gamma(\mathsf{f} : s_1^?, \dots, s_n^? \to s^?) \vdash_{ct} \mathsf{f}(\mathsf{t}_1,\dots,\mathsf{t}_n) : \alpha_0 \bullet \mathcal{C}}\ \text{CT-Fun}$$

$$\frac{\mathcal{C} = \{\alpha_1 =_t s^v\}}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash_{ct} \mathsf{v}() : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Empty}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v, \alpha_2 <:_t s_1^?\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \quad \text{si } \mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v \text{ et } \mathsf{t}_n \neq \mathsf{x*}\ \text{CT-Elem}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_1 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \quad \text{si } \mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v \text{ ou } \mathsf{t}_n = \mathsf{x*}\ \text{CT-Merge}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \qquad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} := \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-Ass} \qquad \frac{\Gamma \vdash \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-BqTerm}$$

où $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$, $\mathsf{v} \in \mathcal{F}^\star$ et α_i sont de variables de type fraîches pour $i \in [1, n]$

FIGURE 6 – Règles d'inférence de types avec sous-typage.

non vide $\mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_n)$, si l'élément t_n est du type du domaine de v et son symbole de tête est différent de v, alors les types α_1 et α tels que $\alpha_1 <:_t s_1^?$ et $\alpha =_t s^v$ sont des types valides respectivement pour t_n et pour la liste originale concaténée avec t_n. Il est encore possible de concaténer deux listes par application de la règle [CT-Merge]. Si ces listes ont un même type s^v, alors un type α tel que $\alpha =_t s^v$ est un type pour la liste résultante. La règle [CT-Merge] est également appliquée pour insérer une variable étoile de type s^v dans une liste avec codomaine s^v.

Notre système d'inférence de types prévoit une règle pour chaque « format » d'expressions Tom, même pour celles qui ne sont pas des termes Tom. Dans la

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\tau =_t \alpha_1, \alpha_1 <:_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathsf{t}_1 \ll |\tau|\ \mathsf{t}_2) : wt \bullet \mathcal{C}}\ \text{CT-Match}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha <:_t \alpha_1, \alpha <:_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathsf{t}_1 \diamond \mathsf{t}_2) : wt \bullet \mathcal{C}}\ \text{CT-Num}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \wedge \ldots \wedge cond_n) : wt \bullet \mathcal{C}}\ \text{CT-Conj}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \vee \ldots \vee cond_n) : wt \bullet \mathcal{C}}\ \text{CT-Disj}$$

$$\frac{\Gamma \vdash_{ct} action_1 : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} action_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (action_1; \ldots; action_n) : wt \bullet \mathcal{C}}\ \text{CT-Action}$$

$$\frac{\Gamma \vdash_{ct} (cond) : wt \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} (action) : wt \bullet \mathcal{C}_2 \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (cond \longrightarrow action) : wt \bullet \mathcal{C}}\ \text{CT-Rule}$$

$$\frac{\Gamma \cup \Gamma_1 \vdash_{ct} rule_1 : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \cup \Gamma_n \vdash_{ct} rule_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \bigcup_{i=1}^{n} \Gamma_i \vdash_{ct} \{rule_1; \ldots; rule_n\} : wt \bullet \mathcal{C}}\ \text{CT-Block}$$

où α_1, α_2 sont de variables de type fraîches

FIGURE 7 – Règles d'inférence de types avec sous-typage – suite.

Figure 7, nous notons que la règle [CT-BLOCK] considère l'union d'un contexte global (commun à tous les jugements de typage dans la prémisse) et de contextes locaux Γ_i pour chaque $rule_i$. Un contexte local Γ_i contient toutes les variables apparaissant dans des termes algébriques *purs* d'un programme Tom, *i*.e. dans les conditions numériques et dans les motifs de conditions de filtrage de $rule_i$ avec

leurs annotations respectives. Ceci évite le besoin de renommer des variables qui ont le même nom mais qui n'apparaissent pas dans les différentes règles $cond \longrightarrow action$. L'union $\Gamma \cup \Gamma_i$ comprend toutes les variables annotées et les opérateurs apparaissant dans une expression $rule_i$.

Les règles d'inférence de type présentées dans les Figures 6 et 7 caractérisent un algorithme d'inférence de types capable de trouver une *solution* pour un jugement de typage basé sur des contraintes.

Définition 21 (Solution pour un jugement de typage basé sur des contraintes)**.** *Soit* Γ *un contexte et* e *une expression* Tom*. Supposons que* $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$*. Une* solution *pour* $(\Gamma, e, \tau, \mathcal{C})$ *est une paire* (σ, τ') *telle que* σ *satisfait* $\mathcal{C}$ *et* $\sigma\tau <:_t \tau'$*, où* $\tau' \in \mathcal{T}y \backslash \mathcal{X}$*.*

L'approche algorithmique du système d'inférence de types est équivalente à l'approche déclarative du système de vérification de types présenté précédemment.

Résolution de contraintes

Les règles d'inférence de types décrivent une manière algorithmique de mapper les types des termes apparaissant dans une expression à un ensemble de contraintes. Lorsque l'on considère du sous-typage, l'ensemble de contraintes obtenu peut être divisé en deux sous-ensembles : un sous-ensemble de contraintes d'égalité $\mathcal{C}_e$ et un sous-ensemble de contraintes de sous-typage $\mathcal{C}_s$. Le premier sous-ensemble est résolu par unification à travers un algorithme solveEqConstraints. Tout d'abord, nous considérons que l'ensemble initial de contraintes d'égalité est en *forme canonique*.

Définition 22 (Forme canonique d'un ensemble de contraintes d'égalité)**.** *On dit qu'un ensemble de contraintes d'égalité* $\mathcal{C}_e$ *est en* forme canonique *s'il vérifie la propriété suivante :*

$$\forall \alpha \in \mathcal{X}, \forall s^? \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\}), \forall v \in \mathcal{F}^\star, (\alpha =_t s^v \in \mathcal{C}_e \Rightarrow \alpha =_t s^? \notin \mathcal{C}_e)$$

Nous présentons dans la Figure 8 un algorithme de résolution de contraintes solveEqConstraints composé d'un ensemble de règles de réécriture conditionnelles. Il produit un ensemble d'erreurs de type *err* possiblement trouvées et une substitution σ qui satisfait un ensemble initial de contraintes d'entrée. En raison des travaux indépendants de Hindley [Hin69] et Milner [Mil78], l'algorithme est basé sur l'idée d'utiliser l'unification pour vérifier qu'un ensemble de contraintes a une solution et, le cas échéant, lui trouver une *solution principale*.

Définition 23 (Solution principale pour un jugement de typage basé sur des contraintes)**.** *Soit* Γ *un contexte,* e *une expression* Tom *et* $\mathcal{C}$ *un ensemble de contraintes composé de* $\mathcal{C}_e$ *et* $\mathcal{C}_s$*. Supposons que* $\Gamma \vdash_{ct} \mathsf{e} : \tau \bullet \mathcal{C}$*. Une* solution principale *pour* $(\Gamma, e, \tau, \mathcal{C})$ *est une solution* (σ, τ') *tel que, lorsque* (σ_1, τ_1) *est aussi une solution pour* $(\Gamma, e, \tau, \mathcal{C})$*, nous avons* $\sigma \lesssim \sigma_1$*.*

Une solution principale (σ, τ) *pour* $(\Gamma, e, \tau, \mathcal{C})$ *est dénommée* type principal *de* e *sous* Γ*.*

(1)	$\tau =_t \tau, err, \sigma$	$\Longrightarrow$	err, σ	
(2)	$s_1^{h_1} =_t s_2^{h_2}, err, \sigma$	$\Longrightarrow$	$\mathsf{isEq}(s_1^{h_1}, s_2^{h_2}) \cup err,\ \sigma$	
(3a)	$s_1^{h_1} =_t \alpha_1, err, \sigma$	$\Longrightarrow$	$err, \downarrow ([\alpha_1 \mapsto s_1^{h_1}] \circ \sigma)$	si $\sigma\alpha_1 = \alpha_1$
(3b)	$s_1^{h_1} =_t \alpha_1, err, \sigma$	$\Longrightarrow$	$\mathsf{isEq}(s_1^{h_1}, s_2^{h_2}) \cup err, \sigma$	si $\sigma\alpha_1 = s_2^{h_2}$
(3c)	$s_1^{h_1} =_t \alpha_1, err, \sigma$	$\Longrightarrow$	$err, \downarrow ([\alpha_2 \mapsto s_1^{h_1}] \circ \sigma)$	si $\sigma\alpha_1 = \alpha_2$
(4a)	$\alpha_1 =_t \alpha_2, err, \sigma$	$\Longrightarrow$	$err, \downarrow ([\alpha_1 \mapsto \alpha_2] \circ \sigma)$	si $\sigma\alpha_1 = \alpha_1 \wedge \sigma\alpha_2 = \alpha_2$
(4b)	$\alpha_1 =_t \alpha_2, err, \sigma$	$\Longrightarrow$	$err, \downarrow ([\alpha_1 \mapsto s_1^{h_1}] \circ \sigma)$	si $\sigma\alpha_1 = \alpha_1 \wedge \sigma\alpha_2 = s_1^{h_1}$
(4c)	$\alpha_1 =_t \alpha_2, err, \sigma$	$\Longrightarrow$	$err, \downarrow ([\alpha_1 \mapsto \alpha_3] \circ \sigma)$	si $\sigma\alpha_1 = \alpha_1 \wedge \sigma\alpha_2 = \alpha_3$
(4d)	$\alpha_1 =_t \alpha_2, err, \sigma$	$\Longrightarrow$	$\mathsf{isEq}(s_1^{h_1}, s_2^{h_2}) \cup err, \sigma$	si $\sigma\alpha_1 = s_1^{h_1} \wedge \sigma\alpha_2 = s_2^{h_2}$
(4e)	$\alpha_1 =_t \alpha_2, err, \sigma$	$\Longrightarrow$	$err, \downarrow ([\alpha_3 \mapsto s_1^{h_1}] \circ \sigma)$	si $\sigma\alpha_1 = s_1^{h_1} \wedge \sigma\alpha_2 = \alpha_3$
(4f)	$\alpha_1 =_t \alpha_2, err, \sigma$	$\Longrightarrow$	$err, \downarrow ([\alpha_3 \mapsto \alpha_4] \circ \sigma)$	si $\sigma\alpha_1 = \alpha_3 \wedge \sigma\alpha_2 = \alpha_4$

où $h_1, h_2 \in \mathcal{F}^\star \cup \{?\}$

FIGURE 8 – Les règles de l'algorithme solveEqConstraints pour résoudre un ensemble de contraintes d'égalité $\mathcal{C}_e$.

Les règles de la Figure 8 sont appliquées sur tous les éléments d'un ensemble de contraintes d'égalité $\mathcal{C}_e$ pour effectuer l'unification. Elles prennent en compte la propriété commutative de l'opérateur $=_t$. Pour une équation de la forme $\tau_1 =_t \tau_2$, chaque instance possible de τ_1 et τ_2 est prise en compte. L'algorithme auxiliaire isEq présenté dans la Figure 9 est appelé chaque fois qu'une équation close est trouvée afin de vérifier si l'égalité des sortes décorées (comme décrit dans la Définition 4) est valable pour τ_1 et τ_2. Dans le cas où la contrainte n'est pas close, la substitution σ est alors incrémentée en fonction des types possiblement mappés à τ_1 et à τ_2 par σ.

Initialement, $\mathcal{C}_e$ n'a que des équations avec deux types τ_1 et τ_2 et la solution σ est vide. Afin de conserver une trace des erreurs de type trouvées lors de la résolution de contraintes, les règles incrémentent un ensemble err – initialement vide – de paires de types pour lesquels l'égalité des sortes décorées n'est pas assurée. Chaque élément de err produit un message d'erreur et indique que $\mathcal{C}_e$ n'a pas de solution, *i*.e. que la solution générée σ n'est pas valide. Cette

$$
\begin{array}{lllll}
\text{(1a)} & s_1^?, s_2^h & \Longrightarrow & \{\} & \text{si } s_1 = s_2 \\
\text{(1b)} & s_1^?, s_2^h & \Longrightarrow & \{(s_1^?, s_2^h)\} & \text{si } s_1 \neq s_2 \\
\\
\text{(2a)} & s_1^h, s_2^? & \Longrightarrow & \{\} & \text{si } s_1 = s_2 \\
\text{(2b)} & s_1^h, s_2^? & \Longrightarrow & \{(s_1^h, s_2^?)\} & \text{si } s_1 \neq s_2 \\
\\
\text{(3a)} & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{\} & \text{si } s_1 = s_2 \wedge v_1 = v_2 \\
\text{(3b)} & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{(s_1^{v_1}, s_2^{v_2})\} & \text{si } s_1 \neq s_2 \vee v_1 \neq v_2
\end{array}
$$

où $h \in \mathcal{F}^\star \cup \{?\}$ et $v_1, v_2 \in \mathcal{F}^\star$

FIGURE 9 – Les règles de l'algorithme isEq pour détecter des erreurs dans un ensemble de contraintes d'égalité $\mathcal{C}_e$.

approche consistant à collecter de paires de types correspond à une approche à la Java car elle permet au système de types de signaler toutes les erreurs de type trouvées au cours de la résolution de contraintes (au lieu de ne signaler que la première erreur). Néanmoins, une autre approche possible serait de mettre fin à la résolution de contraintes lorsque *err* devient non vide pour la première fois, *i*.e. d'ajouter une condition telle que $\mathsf{isEq}(\tau_1, \tau_2) \neq \emptyset$ pour chaque règle.

Le comportement opérationnel de la résolution de contraintes consiste à appliquer progressivement l'algorithme de la Figure 8 sur l'ensemble initial de contraintes $\mathcal{C}_e$ avec un ensemble vide *err* d'erreurs de type et une solution vide σ :

$$\textbf{foldl}\ \mathsf{solveEqConstraints}\ (err, \sigma)\ \mathcal{C}_e$$

L'application de σ à l'équation courante se fait indirectement en conservant σ en *forme saturée*.

Définition 24 (Forme saturée)**.** *Soit* Γ *un contexte,* e *une expression* Tom*,* $\mathcal{C}$ *un ensemble de contraintes composé de* $\mathcal{C}_e$ *et* $\mathcal{C}_s$ *et* σ *une solution pour* $(\Gamma, e, \tau, \mathcal{C})$*. On dit que* σ *est en* forme saturée*, notée* $\downarrow\sigma$*, s'il satisfait la propriété suivante :*

$$\forall \alpha \in \mathcal{X}, (\alpha \in \mathcal{D}om(\sigma) \Rightarrow \alpha \notin \mathcal{R}ange(\sigma))$$

L'algorithme s'arrête lorsqu'il atteint la fin de l'ensemble de contraintes d'égalité. Si $err \neq \emptyset$, alors tous les messages d'erreur collectés pendant la résolution ont été signalés et l'algorithme échoue puisque le σ est rejetée. Sinon, σ est appliquée à l'expression Tom considérée dans le jugement de type basé sur des contraintes.

La notion d'unificateur est appliquée aux ensembles de contraintes en considérant l'égalité des sortes décorées (c.f. Définition 4). Basés sur l'unification, nous définissons une *solution* et une *solution principale* pour un ensemble de contraintes d'égalité $\mathcal{C}_e$.

Définition 25 (Solution pour un ensemble de contraintes d'égalité). *Soit $\mathcal{C}_e$ un ensemble de contraintes composé de contraintes d'égalité $\tau_1 =_t \tau_2$. Une* solution *pour une contrainte d'égalité $\tau_1 =_t \tau_2$ est une substitution σ tel que $\sigma\tau_1 =_t \sigma\tau_1$. Une substitution est une solution pour $\mathcal{C}_e$ si elle est une solution pour chaque contrainte d'égalité dans $\mathcal{C}_e$.*

Définition 26 (Solution principale pour un ensemble de contraintes d'égalité). *Une* solution principale *pour un ensemble de contraintes d'égalité $\mathcal{C}_e$ est une solution σ pour $\mathcal{C}_e$ tel que $\sigma \lesssim \delta$ pour chaque solution δ de $\mathcal{C}_e$.*

Voici la définition de l'algorithme de résolution de contraintes solveEqConstraints.

Définition 27 (Algorithme de résolution des contraintes d'égalité). *L'algorithme* solveEqConstraints *prend en entrée un triplet $(\mathcal{C}_e, err, \sigma)$, où $\mathcal{C}_e$ est un ensemble de contraintes d'égalité, err est un ensemble d'erreurs de type initialement vide et σ est une substitution initialement vide.*

À chaque étape, l'algorithme prend la première équation de $\mathcal{C}_e$ et essaie d'appliquer une des règles de réécriture conditionnelles de la Figure 8 et prend alors la seconde équation pour en faire autant et ainsi de suite jusqu'à ce qu'il atteigne la fin de $\mathcal{C}_e$. Pour une équation entre des types clos et/ou des variables de type mappées à des types clos par σ, la première règle applicable pour la détection d'incompatibilités est appliquée à la contrainte d'égalité. Elle retourne soit un ensemble vide, soit un singleton avec une paire de types. Si l'ensemble résultant est non vide, alors l'algorithme soulève l'erreur de type trouvée et calcule l'union de cet ensemble résultant avec err.

A la fin de $\mathcal{C}_e$, si err est vide, alors la subtitution σ generee constitue une solution pour le $\mathcal{C}_e$ original. Sinon, tous les messages d'erreur collectés pendant la résolution ont été signalés et l'algorithme échoue puisque soit $\mathcal{C}_e$ n'a pas de solution, soit des annotations de type doivent être fournies afin de trouver une solution principale pour le $\mathcal{C}_e$ original.

L'algorithme solveEqConstraints se termine toujours, en échouant lorsqu'un ensemble non-unifiable de contraintes est consideré en entrée ou en retournant une solution principale pour l'ensemble initial des contraintes. En raison de l'utilisation de **foldl**, la terminaison de la résolution de contraintes est équivalent à la terminaison de solveEqConstraints. Cet algorithme est non récursif et correspond à une version dérécursivée de l'algorithme d'unification syntaxique classique, à l'exception de la manipulation d'un ensemble d'erreurs de type *err*. Si aucune erreur n'a été trouvée lors de l'unification (*i.e.* $err = \emptyset$), la substitution est appliquée sur le sous-ensemble de contraintes de sous-typage $\mathcal{C}_s$.

Nous proposons un algorithme de simplification inspiré du travail de Pottier [Pot98b, Pot98a, Pot01] pour la propagation de contraintes de sous-typage. L'agorithme est composé d'une combinaison de trois phases de simplification qui mantiennent σ en forme saturée. La propagation de contraintes de sous-typage est décrite dans ce qui suit et, par la suite, nous présentons une phase

responsable par la génération d'une solution. La dernière phase est celle du ramassage de miettes sur des éventuelles contraintes restantes. Comme dans la résolution des contraintes d'égalité, la solution σ et l'ensemble d'erreurs *err* sont initialement vides. En outre, $\mathcal{C}_s$ne contient que des contraintes d'égalité ou de sous-typage entre des types τ_1 and τ_2. Cependant, un ensemble non vide *err* indique que soit $\mathcal{C}_s$ n'a pas de solution, soit une solution pour $\mathcal{C}_s$ n'a pu être trouvée.

Simplification et clôture

Cette phase effectue une élimination triviale des contraintes de sous-typage réflexives. Elles sont considérées comme dénuées de sens car elles n'ajoutent pas d'informations sur les types qu'elles restreignent. Ainsi, ces contraintes sont supprimées de l'ensemble de contraintes de sous-typage $\mathcal{C}_s$:

$$\{\tau <:_t \tau\} \uplus \mathcal{C}'_s, err, \sigma \quad \Longrightarrow \quad \mathcal{C}'_s, err, \sigma$$

où $\tau \in \mathcal{T}y \backslash \{wt\}$.

L'anti-symétrie est la propriété suivante de l'ordre partiel $<:_t$ à être considérée. Lorsque cela est possible, nous essayons de réécrire des contraintes de sous-typage en des contraintes d'égalité afin de résoudre un plus grand nombre de contraintes par unification. À cet effet, nous considérons la règle de simplification suivante :

$$\{\tau_1 <:_t \tau_2, \tau_2 <:_t \tau_1\} \uplus \mathcal{C}'_s, err, \sigma \quad \Longrightarrow \quad \mathcal{C}'_s, \mathsf{solveEqConstraints}(\{\tau_1 =_t \tau_2\}, err, \sigma)$$

Nous rappelons que l'algorithme solveEqConstraints retourne l'ensemble *err* et la solution σ générés.

Les variables de type qui apparaissent dans un ensemble de contraintes de sous-typage peuvent être indirectement restreintes en raison de la propriété transitive de $<:_t$. Ainsi, des contraintes résultantes sont calculées par l'application de la clôture transitive de $<:_t$ qui maintient également les contraintes existantes de $\mathcal{C}_s$:

$$\{\tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \uplus \mathcal{C}'_s, err, \sigma \quad \Longrightarrow \quad \{\tau_1 <:_t \tau_2, \tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \cup \mathcal{C}'_s, err, \sigma$$

où $\tau_1, \tau_2 \in \mathcal{T}y \backslash \{wt\}$ et $\alpha \in \mathcal{X}$.

La clôture transitive de $<:_t$ sur $\mathcal{C}_s$ représente la *forme close* d'un ensemble de contraintes de sous-typage qui est stable via transitivité.

Définition 28 (Forme close)**.** *Soit $\mathcal{C}_s$ un ensemble de contraintes de sous-typage. On dit que $\mathcal{C}_s$ est* clos pour la transitivité, *ou* clos *pour faire plus court, si et seulement si lorsque $\{\tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \uplus \mathcal{C}_s$ tel que $\tau_1, \tau_2 \in \mathcal{T}y \backslash \{wt\}$ et $\alpha \in \mathcal{X}$, alors $\{\tau_1 <:_t \tau_2\}$ est défini et inclus dans $\mathcal{C}_s$.*

Détection d'incompatibilités

Puisque la hiérarchie entre les types des termes est fournie par la clôture transitive de $<:_t$ sur un contexte Γ, nous définissons une règle pour valider les contraintes de sous-typage closes. L'objectif est de détecter quelles sont les contraintes closes de $\mathcal{C}_s$ qui ne sont pas valides en ce qui concerne la hiérarchie de types considérée et, par conséquent, de simplifier l'ensemble des contraintes et peut-être signaler des messages d'erreur.

$$\{s_1^{h_1} <:_t s_2^{h_2}\} \uplus \mathcal{C}'_s, err, \sigma \quad \Longrightarrow \quad \mathcal{C}'_s, \mathsf{isSub}(s_1^{h_1}, s_2^{h_2}) \cup err, \sigma$$

où $h_1, h_2 \in \mathcal{F}^\star \cup \{?\}$.

Dans cette phase, l'algorithme isSub, présenté dans la Figure 10, est appelé à chaque fois qu'une contrainte de sous-typage close est trouvée dans $\mathcal{C}_s$. Les règles pour la détection d'erreurs sont basées sur la définition de l'ordre partiel $<:_t$ (voir Définition 5) en considérant la hiérarchie de types fournie par Γ^*.

$$\begin{array}{lllll}
\text{(1a)} & s_1^?, s_2^? & \Longrightarrow & \{\} & \text{if } (s_1^? <:_t s_2^?) \in \Gamma^* \\
\text{(1b)} & s_1^?, s_2^? & \Longrightarrow & \{(s_1^?, s_2^?)\} & \text{if } (s_1^? <:_t s_2^?) \notin \Gamma^* \\
\\
\text{(2)} & s_1^?, s_2^v & \Longrightarrow & \{(s_1^?, s_2^v)\} & \\
\\
\text{(3a)} & s_1^v, s_2^? & \Longrightarrow & \{\} & \text{if } (s_1^? <:_t s_2^?) \in \Gamma^* \\
\text{(3b)} & s_1^v, s_2^? & \Longrightarrow & \{(s_1^v, s_2^?)\} & \text{if } (s_1^? <:_t s_2^?) \notin \Gamma^* \\
\\
\text{(4a)} & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{\} & \text{if } (s_1^? <:_t s_2^?) \in \Gamma^* \wedge v_1 = v_2 \\
\text{(4b)} & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{(s_1^{v_1}, s_2^{v_2})\} & \text{if } (s_1^? <:_t s_2^?) \notin \Gamma^* \vee v_1 \neq v_2
\end{array}$$

où $v, v_1, v_2 \in \mathcal{F}^\star$

FIGURE 10 – Les règles de l'algorithme isSub pour détecter des erreurs dans un ensemble de contraintes de sous-typage $\mathcal{C}_s$.

Canonisation

Cette phase vise à réduire l'espace de recherche de la résolution des contraintes de sous-typage. L'idée est de mettre l'ensemble de contrainte de sous-typage en *forme canonique*, *i*.e. de limiter chaque variable de type apparaissant dans cet ensemble par au plus un type clos par borne.

Définition 29 (Forme canonique d'un ensemble de contraintes de sous-typage)**.** *Soit $\mathcal{C}$ un ensemble de contraintes de sous-typage. On dit que $\mathcal{C}$ est en* forme canonique *si et seulement si chaque variable de type α a au maximum un type clos comme majorant et un type clos comme minorant.*

Afin de mettre un ensemble arbitraire de contraintes de sous-typage en forme canonique, nous considérons les règles suivantes :

$$\begin{array}{lcl}
\{s_1^{h_1} <:_t \alpha, s_2^{h_2} <:_t \alpha\} \uplus \mathcal{C}'_s, err, \sigma & \Longrightarrow & \{s^? <:_t \alpha\} \cup \mathcal{C}'_s, err, \sigma \\
 & & \text{si } \exists s^? \text{ tel que} \\
 & & s^? \text{ est la borne supérieure de } \{s_1^{h_1}, s_2^{h_2}\} \\
\{s_1^{h_1} <:_t \alpha, s_2^{h_2} <:_t \alpha\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \mathcal{C}', \{(s_1^{h_1}, s_2^{h_2})\} \cup err, \sigma \\
 & & \text{si } \nexists s^? \text{ tel que} \\
 & & s^? \text{ est la borne supérieure de } \{s_1^{h_1}, s_2^{h_2}\} \\
 & & \\
\{\alpha <:_t s_1^{h_1}, \alpha <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \{\alpha <:_t s_1^{h_1}\} \cup \mathcal{C}', err, \sigma \\
 & & \text{si } \Gamma^*(s_1^? <:_t s_2^?) \wedge (h_1 = h_2 \vee h_2 =?) \\
\{\alpha <:_t s_1^{h_1}, \alpha <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \{\alpha <:_t s_2^{h_2}\} \cup \mathcal{C}', err, \sigma \\
 & & \text{si } \Gamma^*(s_2^? <:_t s_1^?) \wedge (h_1 = h_2 \vee h_1 =?) \\
\{\alpha <:_t s_1^{h_1}, \alpha <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \mathcal{C}', \{s_1^{h_1}, s_2^{h_2}\} \cup err, \sigma \\
 & & \text{si } s_1^{h_1} \text{ et } s_2^{h_2} \text{ ne sont pas comparables}
\end{array}$$

où $h_1, h_2 \in \mathcal{F}^\star \cup \{?\}$ et $\alpha \in \mathcal{X}$.

Les erreurs provoquées par des types incompatibles peuvent également être signalées au cours du processus de canonisation. Cela se produit lorsque deux minorants (resp. majorants) ne sont pas comparables ce qui génère une nouvelle paire (type,type) ajoutée à *err*.

Génération d'une solution

On dit que l'ensemble de contraintes de sous-typage $\mathcal{C}_s$ résultant de la séquence de phases de simplification, de clôture, de détection d'incompatibilités et de canonisation (*i*.e. le processus de propagation) est en *forme résolue* si aucune erreur n'a été trouvée, c'est-à-dire $err = \{\}$. Par la suite, nous nous intéressons à trouver une *solution* pour $\mathcal{C}_s$.

Définition 30 (Solution pour un ensemble de contraintes de sous-typage)**.** *Soit $\mathcal{C}_s$ un ensemble de contraintes composé de contraintes de sous-typage $\tau_1 <:_t \tau_2$. Une* solution *pour une contrainte de sous-typage $\tau_1 <:_t \tau_2$ est une substitution σ telle que $\sigma\tau_1 <:_t \sigma\tau_2$. Une substitution σ est solution de $\mathcal{C}$ si elle est solution de toutes les contraintes de sous-typage dans $\mathcal{C}_s$.*

Lors du processus de résolution de $\mathcal{C}_s$ et après chaque application d'une règle pour la génération d'une solution, nous voulons nous assurer que l'ensemble de contraintes courant est satisfaisable, de manière à détecter des erreurs le plus tôt possible. Par conséquent, nous combinons les règles relatives à la propagation de contraintes avec les règles suivantes destinées à la résolution de contraintes :

$$\{\alpha <:_t s^h\} \uplus \mathcal{C}'_s, err, \sigma \quad \Longrightarrow \quad [\alpha \mapsto s^h]\mathcal{C}'_s, err, \{\alpha \mapsto s^h\} \circ \sigma$$

$$\{s^h <:_t \alpha\} \uplus \mathcal{C}'_s, err, \sigma \quad \Longrightarrow \quad [\alpha \mapsto s^h]\mathcal{C}'_s, err, \{\alpha \mapsto s^h\} \circ \sigma$$

où $h \in \mathcal{F}^\star \cup \{?\}$ et $\alpha \in \mathcal{X}$.

Au lieu d'appliquer ces règles de manière exhaustive, seule la première règle applicable est appliquée. Ainsi, l'ordre des règles est vraiment important afin de permettre le choix du type le moins restrictif lorsque deux possibilités existent. Par exemple, dans le cas où une variable de type α est limitée par $\alpha <:_t s_1^{h_1}$ et $s_2^{h_2} <:_t \alpha$ dans $\mathcal{C}_s$, le type $s_1^{h_1}$ sera assigné à α. Considérant un système de types pour un langage embarqué, cette approche vise à être compatible avec le système de types de Java où le transtypage ascendant est implicite et sûr. Cela réduit les conflits avec la vérification de types de Java qui vérifie chaque transtypage dans le code généré par Tom.

Ramassage de miettes

Beaucoup de variables de type sont créées lors de la génération des contraintes de type par l'algorithme d'inférence de types. Toutefois, certaines de ces variables de type ne représentent pas des types de termes, mais sont utilisées pour restreindre indirectement les types de termes d'entrée. Nous définissons un ensemble de *types d'entrée* $\mathcal{I}$ afin de distinguer les deux genres de variables de type.

Définition 31 (Types d'entrée d'un jugement de typage)**.** *Considérons un jugement de typage* $\varphi = \Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$*. L'ensemble de* types d'entrée *de* φ*, noté* $\mathcal{I}$*, est un ensemble qui satisfait les propriétés suivantes :*

- *si* $e = \{rule_1; \ldots; rule_n\}$*, alors* $\forall i \in [1,n], \forall \alpha \in \mathcal{I}, \exists \mathsf{x} \in \mathcal{V}, (\ \mathsf{x} : \alpha \in \Gamma'_i)$ *où* $\Gamma = \Gamma' \bigcup_{i=1}^{n} \Gamma'_i$
- *si* $e \neq \{rule_1; \ldots; rule_n\}$*, alors* $\mathcal{I} = \{\}$

Bien qu'étant utiles pour le calcul de la clôture de $\mathcal{C}_s$, les contraintes limitent seulement les variables de type qui n'appartiennent pas à $\mathcal{I}$ et qui ne sont pas mappées à de types d'entrée ne sont plus nécessaires une fois que la phase de propagation est terminée. Ainsi, ces contraintes sont éliminées par les règles suivantes :

$$\{\alpha_1 <:_t \alpha_2\} \uplus \mathcal{C}'_s, err, \sigma, \mathcal{I} \implies \mathcal{C}'_s, err, \sigma, \mathcal{I} \qquad \text{si } \alpha_1, \alpha_2 \notin \sigma\mathcal{I}$$

$$\{\alpha_1 <:_t \alpha_2\} \uplus \mathcal{C}'_s, err, \sigma, \mathcal{I} \implies \mathcal{C}'_s, \{(\alpha_1, \alpha_2)\} \cup err, \sigma, \mathcal{I} \qquad \text{si } \alpha_1 \in \sigma\mathcal{I} \vee \alpha_2 \in \sigma\mathcal{I}$$

où $\alpha_1, \alpha_2 \in \mathcal{X}$.

Le ramassage de miettes consiste à vérifier que toutes les variables de type d'entrée sont dans le domaine de la solution σ. Cela signifie que toutes les variables qui apparaissent dans les conditions numériques et dans les motifs des conditions de filtrage ont leur type inféré. Ces expressions correspondent à des termes algébriques *purs* et leurs variables sont, par conséquent, des variables Tom *pures* contrairement aux variables qui ont été éventuellement déclarées dans du code Java.

Voici la définition de l'algorithme de propagation et de résolution des contraintes de sous-typage solveSubConstraints.

Définition 32 (Algorithme de résolution des contraintes de sous-typage)**.** *L'algorithme* solveSubConstraints *prend en entrée un quadruplet* $(\mathcal{C}_s, err, \sigma, \mathcal{I})$*, où* $\mathcal{C}$ *est un ensemble de contraintes de sous-typage,* err *est un ensemble d'erreurs de type initialement vide,* σ *est une substitution initialement vide et* $\mathcal{I}$ *est un ensemble de variables de type.*

L'algorithme applique récursivement les règles de la phase de simplification et de clôture jusqu'à ce qu'un point fixe soit atteint. Ensuite, il boucle sur $\mathcal{C}_s$ *jusqu'à ce qu'aucune des règles ne soit applicable à* $\mathcal{C}_s$*. La phase de détection d'incompatibilités est ensuite initialisée au début de cette boucle. Si l'ensemble* err *résultant est non vide, alors l'algorithme s'arrête et signale toutes les erreurs de type trouvées. Sinon, la phase de canonisation est déclanchée.*

Ensuite, l'algorithme essaie d'appliquer une des règles de génération de solution. Si au moins l'une d'entre elles est applicable, alors celle-ci est appliquée à $\mathcal{C}_s$ *et l'algorithme reprend au début de la boucle. Sinon, l'algorithme interrompt la boucle.*

Une fois sorti de la boucle, le ramassage de miettes est activé jusqu'à ce que $\mathcal{C}_s$ *devienne vide. À ce moment, si* err *est vide, alors la substitution* σ *générée constitue une solution pour l'ensemble* $\mathcal{C}_s$ *original. Sinon, tous les messages d'erreur collectés pendant la résolution ont été signalés et l'algorithme échoue puisque soit* $\mathcal{C}_s$ *n'a pas de solution, soit des annotations de type doivent être fournies afin de trouver une solution pour le* $\mathcal{C}_e$ *original.*

Nous notons que l'application de l'une des règles de génération conserve $\mathcal{C}_s$ dans sa forme close et ne génère ni de contraintes réflexives ni de paires de contraintes symétriques. Pour cette raison, la phase de simplification et de clôture n'est pas incluse dans la boucle.

Puisque l'inférence de types génère à la fois un ensemble de contraintes d'égalité $\mathcal{C}_e$ et un ensemble de contraintes de sous-typage $\mathcal{C}_s$, la résolution de contraintes est implémentée par l'algorithme solveConstraints qui est constitué d'une combinaison de solveEqConstraints et solveSubConstraints. Initialement, l'ensemble de contraintes d'égalité $\mathcal{C}_e$ et les ensembles vides err et σ sont passés en paramètres à solveEqConstraints. Puis les ensembles retournés err et σ en forme saturée sont vérifiés. Si $err \neq \emptyset$, alors tous les messages d'erreur collectés au cours de la résolution des contraintes d'égalité ont été signalés et l'algorithme solveConstraints échoue puisque σ est rejetée. Sinon, σ est appliquée à l'ensemble de contraintes de sous-typage $\mathcal{C}_s$. L'ensemble $\mathcal{C}'_s$ obtenu à partir de l'application de σ est passé en paramètre à solveSubConstraints, conjointement avec err, σ et un ensemble $\mathcal{I}$ de variables de type d'entrée. L'algorithme s'arrête lorsque l'ensemble de contraintes devient vide. Si $err \neq \emptyset$, alors l'algorithme échoue et σ est rejetée. Sinon, σ est appliquée à l'expression Tom considérée dans le jugement de type basé sur des contraintes.

L'algorithme de solveConstraints se termine toujours, en échouant lorsqu'un ensemble non-unifiable de contraintes est consideré en entrée ou en retournant une solution pour l'ensemble de contraintes initial.

Conclusion

Les systèmes de types sont appliqués à la programmation informatique en tant que méthode formelle capable de classer les termes selon les genres (*i*.e. les sortes) de valeurs calculées à l'exécution. Plutôt que de prouver l'absence de résultats calculés inattendus, ces systèmes visent à prouver l'absence de termes dénués de sens par rapport à une spécification de types. Cette thèse se situe dans le cadre de la réécriture de termes intégrée dans la programmation orientée objet. Son but est de développer un système de types sûr compatible avec celui du langage de programmation hôte. En particulier, il dispose de sous-typage pour le support de filtrage de motifs associatif sur les types de données algébriques avec constructeurs variadiques. Dans ce travail, nous avons spécifiquement considéré le langage de réécriture Tom et son environnement développé dans l'équipe-projet Pareo.

Notre premier objectif était d'améliorer la sûreté du langage Tom à l'aide de vérification et d'inférence de types en tant qu'approches d'analyse appliquées par le mécanisme de typage. Nous avons ensuite suivi l'idée de John Mitchell [Mit84] à propos de l'inférence de type comme une forme de vérification de type et nous avons donc inclus des contraintes dans les jugements de typage. Une difficulté préliminaire se pose sur la définition d'un système basé sur des contraintes dont les éléments constitutifs (*i*.e. les contraintes d'égalité) sont résolus par unification et les types sont interprétés comme des termes non sortés. Ces types suffisent pour distinguer les termes multi-sortés construits à partir d'opérateurs variadiques identiques. Nous avons résolu cette difficulté par la définition de types comme des sortes de base décorées avec des symboles de fonctions variadiques. Nous avons également introduit des variables de type pour représenter des types non interprétés du système d'inférence de types à base de contraintes, comme le suggère Benjamin Pierce [Pie02]. Son travail a inspiré la description du système de vérification de types comme un calcul de la preuve pour les jugements de typage sur des expressions Tom annotées avec des types. Une autre contribution a été la formalisation de la sémantique opérationnelle à grand pas du langage Tom selon à la fois le formalisme définit par Gilles Kahn [Kah87] et le travail de Didier Rémy [Ré02]. Cela fut utile pour prouver la sûreté de notre système de types.

L'amélioration de l'expressivité du filtrage de motifs fourni par Tom était notre principal but. À cet égard, nous avons eu l'intention de permettre l'inclusion de types au niveau du motif, ce qui a mené à une compatibilité de types plus permissive. Inspirés par le travail de Luca Cardelli [Car88], nous avons décidé d'un système de types nominal considérant le sous-typage comme un ordre partiel sur les types. Au lieu de ne considérer que les contraintes d'égalité, le système d'inférence de types génère un ensemble de contraintes d'égalité et de sous-typage (*i*.e. d'inéquation) devant être simplifié et résolu par la suite. Une difficulté majeure était de s'assurer que toutes les combinaisons de contraintes soient traitées par les règles de réécriture de la résolution de contraintes de sous-typage. Notre point de départ pour résoudre ce problème a été l'algorithme de

simplification donné par François Pottier [Pot01]. Nous avons divisé la phase de simplification, tout comme lui, en sous-phases responsables de la production d'une représentation simplifiée de toutes les solutions d'un ensemble de contraintes. Toutefois, contrairement à Pottier et bien que nous ayons ajouté des règles pour la génération d'une seule solution, nous n'avons pas exhibé de classe d'ensembles de contraintes qui ont des solutions. Cependant, nous avons obtenu des résultats théoriques intéressants comme la preuve de l'équivalence entre l'approche algorithmique du système d'inférence de types et l'approche déclarative du système de vérification de types enrichi avec une règle de subsomption. La formalisation de la sémantique opérationnelle du langage Tom a été étendue pour supporter le sous-typage et permettre la preuve de la sûreté du système de types actuel.

D'un point de vue pratique, après une phase de prototypage réussie, nous nous sommes concentrés sur l'implémentation du système d'inférence de types à base de contraintes d'égalité en Tom et dans Tom. Le système Tom possède une structure en pipeline où l'arbre de syntaxe abstraite (AST) représentant le programme d'entrée Tom est séquentiellement transformé par chaque phase de compilation. Cela a soulevé une difficulté initiale relative à l'identification du sous-ensemble des nœuds de l'AST devant être considéré par les règles de typage. Bien que l'ajout de nouveaux constructeurs de données pour la représentation des contraintes de type a été sans impact pour les autres phases, le code Tom a été remanié de façon à nettoyer et à relocaliser les fonctionnalités qui ne concernent pas le typeur. Ce qui a été implémenté dans le compilateur Tom correspond exactement à une version dérécursivée de l'algorithme classique d'unification syntaxique et aux règles d'inférence de type décrites dans cette thèse. Cela donne tout d'abord un environnement théorique solide et éprouvé à la phase de typage du système Tom. En outre, cette correspondance forte entre l'implémentation et la théorie nous permet d'atteindre les objectifs de renforcement de la sûreté du système Tom avec les différentes preuves (de correction, de complétude, de préservation, de progrès, de terminaison) que nous avons écrites. Dans l'ensemble, nous pouvons dire que nous avons obtenu une compatibilité de types plus permissive à la fois en théorie et en pratique pour des constructions de filtrage de Tom tout en préservant la sûreté du typage.

Le coût supplémentaire de ces caractéristiques n'est pas excessif. Les expérimentations ont certes montré que les temps d'inférence de types et de compilation totale des programmes Tom peuvent augmenter pour des codes de grande taille. Cependant, nous pensons que le temps reste dans des limites acceptables. Cette augmentation est en fait plutôt attendue puisque notre typeur effectue l'inférence de types pour un plus grand nombre de constructions Tom par rapport au typeur qui existait auparavant. Néanmoins, nous avons essayé de maintenir le temps additionnel aussi bas que possible grâce à quelques optimisations peu profondes du typeur, notamment en ce qui concerne le nombre de variables de type et de contraintes générées par les règles d'inférence de type. Il nous paraît raisonnable de supposer qu'une extension de nos travaux pourrait apporter des nouvelles optimisations plus profondes permettant notamment

un gain de temps. En particulier, nous pensons qu'un examen en profondeur de l'impact des stratégies dans l'implémentation du typeur pourrait contribuer à réduire le temps de typage. Une revue des structures de données utilisées pourrait également abaisser le temps additionnel de la phase de typage. Enfin, l'algorithme de propagation et de résolution des contraintes de sous-typage nous semble aussi pouvoir bénéficier d'optimisations.

Bien que n'étant pas directement lié à notre système de types, le support de sous-typage par l'outil Gom constitue une facilité souhaitable pour la définition des ancrages organisés dans une structure d'héritage. Cela permettrait l'utilisation standard de Tom dans les cas où les ancrages sont générés automatiquement par Gom. Par conséquent, la hiérarchie de types définie par les relations de sous-typage entre les types de données algébriques serait automatiquement préservée par les ancrages et les types de données concrètes qui les implémentent. Nous envisageons également d'assurer une utilisation plus directe du sous-typage dans du filtrage de motifs à travers l'introduction de *motifs typés* dans Tom. Ces motifs pourraient avoir la forme `x : t` tel que `x` est une variable du motif et `t` est un motif de type algébrique. L'annotation de type explicite de l'opérateur « << » ne serait alors plus nécessaire puisqu'une condition de filtrage (*i*.e. une équation de filtrage) `x << t subj` est extensionnellement équivalente à `x : t << subj`. Ainsi, un motif typé filtrerait tout sujet dont la valeur a le même type que le motif et pourrait apparaitre même dans les arguments de fonctions.

D'un point de vue théorique, des améliorations intéressantes restent à faire dans notre système de types basé sur des contraintes de sous-typage. On pourrait imaginer une nouvelle approche de ramassage de miettes basée sur des *polarités*. À travers l'adaptation de l'idée de François Pottier [Pot01], nous devrions annoter toutes les variables de type avec des signes - et + représentant respectivement un type d'une variable algébrique et un type d'une variable concrète (i.e. non algébrique). Cela permettrait l'élimination des contraintes superflues avant la phase de génération d'une solution. En outre, nous espérons réussir à prouver des propriétés relatives à la fiabilité de notre système de types. Nous avons particulièrement l'intention de montrer deux résultats théoriques : *a*) notre mécanisme de typage dérive les types les moins restrictifs qui satisfont toutes les contraintes générées, et *b*) la définition de la classe d'ensembles de contraintes pour laquelle notre algorithme de résolution de contraintes est capable de trouver une solution. Au demeurant, une amélioration naturelle de notre système de types est la manipulation de nouvelles caractéristiques d'expressivité disponibles au sein des langages dans lesquels Tom peut être embarqué. Par exemple, nous pourrions souligner le *polymorphisme paramétrique* comme un concept très utile aux paradigmes de programmation fonctionnelle et orientée objet. En conséquence, cela permettrait aux ancrages et méthodes usuels qui incluent des constructions de filtrage de manipuler des termes algébriques de types différents dans Tom.

Plan de la thèse

La suite du manuscrit, en langue anglaise, est composé de cinq chapitres. Le premier chapitre introduit les définitions fondamentales des langages basés sur la réécriture. Nous présentons ensuite le langage de réécriture Tom dans le second chapitre. Les deux chapitres suivants concernent le système de types proposé pour Tom. Une discussion sur son utilisation et ses avantages est présentée dans le dernier chapitre. Tous les chapitres sont ici résumés brièvement :

Le Chapitre 1 présente les notions préliminaires utiles pour une meilleure compréhension des chapitres suivants : l'algèbre de termes, les théories équationnelles et la réécriture de termes.

Le Chapitre 2 fournit une présentation informelle du langage Tom comme le principal support de ce travail. Il explique la fonctionnalité et l'utilisation des principales constructions Tom pour : la définition de signatures algébriques représentant des objets concrets, la définition de règles de réécriture par le biais de filtrage de motifs associatif sur des termes algébriques, la simplification des définitions des motifs, la définition de stratégies élémentaires, et l'application de stratégies pour contrôler des règles de réécriture. Il présente également le processus de compilation des programmes Tom et un résumé des principales contributions de cette thèse dans le cadre du développement de Tom.

Le Chapitre 3 introduit une importante contribution de cette thèse pour la sûreté du langage Tom. Il formalise les termes algébriques Tom et la sémantique opérationnelle de la syntaxe Tom. Il propose également une notion originelle de types et un système de vérification de types avec support des termes algébriques construits sur des opérateurs variadiques. Le typage est ensuite prouvé correct par rapport à la sémantique et preservé par évaluation. Un système d'inférence de types à base de contraintes est une autre proposition de ce chapitre. Nous montrons que l'algorithme de résolution de contraintes termine et que le système d'inférence de types est complet et correct par rapport au système de vérification de types. Enfin, les aspects d'implémentation et d'intégration du système d'inférence de types sont présentés.

Le Chapitre 4 introduit l'objectif principal de cette thèse qui consiste en l'amélioration de l'expressivité du langage Tom. Il étend la sémantique opérationnelle de Tom pour permettre l'inclusion de types au niveau du motif. Le système de types précédent est enrichi avec la possibilité de déclarations de sous-typage, bien que l'héritage multiple et la surcharge d'opérateurs soient interdits. Cet affaiblissement de la notion de compatibilité de types est également pris en compte par le système d'inférence de types basé sur des contraintes de sous-typage proposé dans ce chapitre. Par conséquent, un algorithme de résolution de contraintes de sous-typage est décrit comme une combinaison de propagation de contraintes, de génération d'une solution et de ramassage de miettes. Nous montrons la termi-

naison de cet algorithme ainsi que les propriétés précédentes de sûreté, de correction et de complétude du système de types. Le système d'inférence de types est ensuite décrit d'un point de vue pratique concernant des structures de données, des algorithmes et des stratégies requises pour son implémentation.

Le Chapitre 5 illustre les avantages apportés par l'utilisation du sous-typage lors de la programmation en Tom. Cela favorise le système de types basé sur des contraintes de sous-typage dont les règles de typage couvrent un plus grand nombre de constructions Tom par rapport au système de types disponible au début de cette thèse. Nous considérons certains programmes Tom afin d'étalonner la performance du système de types décrit formellement dans les deux chapitres précédents. Enfin, nous présentons une étude de cas de transformation de modèles à l'aide de Tom, ce qui constitue une application réelle du sous-typage en Tom.

Introduction

In computer programming, formal methods have been developed for helping guarantee that the behavior of a system corresponds exactly to its programmer's expectation. The desired behavior must be provided through a specification to be subsequently proved in order to ensure that the system behaves as expected. However, specifications need to be (at least partially) manually written and demand a good knowledge on the adopted logic in order to guide and control the process of proving. For that reason, other well-known and less general formal approaches such as *type systems* are often considered. Type sytems aim to prevent the occurence, not of unexpected computed results, but of meaningless terms in regard to a type specification.

The general goal of this thesis is to define such a type system within the scope of the development of the Tom programming language [BBK+07, BBB+11]. In particular, this type system features subtyping for support of associative pattern matching on algebraic data types with variadic constructors.

In the rest of this introduction we present a quick survey of the main subjects tackled in this thesis before outlining the contents of each chapter. We begin with a brief discussion about type systems as a formal approach for proving the absence of ill-typed terms.

Type systems

Type systems constitute an inference system for classifying terms according to the kinds (i.e. types) of values they compute when executed. A type system is expected to be safe : a program classified as well-typed at compile time must avoid execution errors due to "semantically empty" instructions. For instance, Peano's successor function suc takes an argument of type nat and returns a result of type nat as indicated by its rank suc : nat → nat. Therefore, the instruction suc(0) is considered well-typed while a type incompatibility is detected for the instruction suc(-1) since -1 is not a natural number.

A type system, seen as an inference system, is composed of (typing) inference rules whose application depends on the kind of verification to be performed :

- type checking, for which the typing rules describe a declarative approach to verify statically if all type annotated terms of a program are well-typed,
- type inference, for which the typing rules describe an algorithmic approach to infer statically the type of all terms that are not type annotated in a

program.

Both approaches can be shown to be equivalent through the demonstration of two properties : soundness, which states that every typing judgment that can be derived from the typing rules for type inference also follows from those for type checking ; and completeness, which ensures that a solution validated by the typing rules for type checking can be extended to a solution proposed by the typing rules for type inference.

John Mitchell [Mit84] defines type inference as a form of type checking. Type inference systems are usually based on type constraints following the original idea of Roger Hindley [Hin69]. He proposed a type inference algorithm to generate systems of equality constraints and solve them. This work was later independently developed by Robin Milner [Mil78] who extended it to support the abstraction of a base type with respect to a type variable. He suggested the resolution of type constraints by *unification* which consists in finding a substitution that satisfies all equalities. Milner also proved the soundness and correctness of the type system. The main advantage of the constraint-based approach is that the addition of new kind of constraints only requires the modification of the typing rules and the algorithm for resolution of constraints.

In this thesis we are interested in a safe type system for algebraic terms built from operators with fixed and varying arities. We present a notion of types including enough information for the distinction of different variadic operators. We focus on a type checking system and a constraint-based type inference system featuring *subtyping*. Additionally, we would like this system to detect as many type errors as possible in one of the earliest phases of the language compiler.

Subtyping

The introduction of subtyping into type systems enhances their expressiveness since it leads to a more permissive type compatibility. Generalizations of the Hindley/Milner system with subtyping have been proposed by several authors, such as Luca Cardelli [Car88] and John Mitchell [Mit84]. The latter considered *structural subtyping* where subtype relationships are determined structurally in function of the arities of the type constructors. Differently, Cardelli proposed the notion of *nominal subtyping* as a partial order over types. This involves the addition of a new typing rule stating that if a type τ_1 is a subtype of τ_2 then every term of type τ_1 is of type τ_2 as well. As a result, well-behaved programs which were previously considered as ill-typed may now be accepted. In this sense, types of variables and functions symbols occurring in a term do not need to match exactly the type of its type signature, but instead only be subtypes of it.

In the presence of subtyping, type inference requires to extend the constraint language with subtyping constraints in addition to the equality ones. Therefore, instead of solving a set of equations, the type inference problem re-

duces to simplifying and solving a set of inequalities. Subtyping-constraint-based types systems have been thoroughly studied. Inspired by the work of Mitchell, You-Chin Fuh and Prateek Mishra [FM88] focused on computing subtype relationships between base types and determining whether the constraint set is solvable. However, the idea of separating the generic inference system from the details concerning constraint simplification and resolution was first developed by Stefan Kaes [Kae92] and later by Valery Trifonov and Scott Smith [TS96].

In this thesis we concentrate on the definition of both a type system featuring nominal subtyping and a solving algorithm for equality and subtyping constraints. Indeed, some authors addressed the problem of simplifying (large) constraint sets. Alexander Aiken and Edward Wimmers [AW92] proposed a powerful but complex inference algorithm which performs transformation of systems of constraints while preserving the set of solutions. Besides Trifonov and Smith, François Pottier [Pot01] also presented a simpler approach to simplify constraints using the notion of *constraint entailment*, i.e. constraint simplification which is solution preserving. His type system supports recursive types and all base types are organized in a lattice. Moreover, Pottier described a constraint simplification algorithm made up of several complementary algorithms. Although dealing with a different notion of types, his approach inspired our constraint resolution algorithm, especially the *canonization* sub-algorithm. However, in addition to constraint simplification, we concentrate on the generation of a single solution as a mapping from type variables to base types.

The rules of our type inference system describe an algorithmic way to map types of terms occurring in an expression to a constraint set. The constraint set obtained is divided into an equality constraint set and a subtyping constraint set. For the first set, we present a derecursivated version of the classical syntactic unification algorithm used in Hindley/Milner's type system. The propagation of subtyping constraints is done by the combination of three simplification phases. *Simplification and closure* eliminates reflexive subtyping constraints and performs the transitive closure on type variables of the resulting constraint set. *Incompatibility detection* searches for subtyping constraints that are not valid with respect to the type specification. Finally, *canonization* ensures that each type variable occurring in the constraint set has at most one upper bound and one lower bound. Constraint propagation is combined with constraint resolution. Rules for *generation of solution* are applied in order to obtain a solution with least restrictive types. Then, *garbage collection* determines which constraints are unnecessary and eliminates them. The whole process of constraint resolution is shown to be terminating.

The type system with subtyping conceived in this thesis is dedicated to the support of *associative pattern matching* on algebraic terms representing objects. As an essential feature of the object-oriented paradigm, inheritance determines subtype relationships between objects that are seen as algebraic terms. Accordingly, these terms can be of different subtypes but still uniformly manipulated as if belonging to a common supertype.

Associative pattern matching with subtyping

The notions of algebraic types and terms are often not present in the majority of object-oriented languages. Moreover, these languages do not provide pattern matching constructs which constitute the core of functional and rewriting-based languages. Pattern matching is an operation to check whether a given pattern occurs in the input data structures. If it does, then the variables occurring in the pattern are instantiated according to the solution of the match-equation and the relevant statement is executed in the environment extended by this solution.

Nowadays, a variety of languages feature pattern matching such as Maude [CDE+07], Scala [Ode11, Cre06], ELAN [BKK+98, BCD+06], ... One of them is Tom which is the target implementation language of this thesis. Its particularity is to focus on allowing a host language such as Java to be extended to support pattern matching constructs.

We can use Tom to define an algebraic type nat for Peano integers and their addition operation :

```
nat = zero() | suc(n:nat)
...
public nat Plus(nat t1, nat t2) {
  %match(t1, t2) {
    zero(), x -> { return `x; }
    suc(y), x -> { return `suc(Plus(y,x)); }
  }
}
```

The **%match** construct implements the notion of pattern matching in the Java method Plus. This method takes two terms t1 and t2 of algebraic type nat representing the Peano integers and return the sum of them which is also of type nat :

- if t1 = zero() and t2 = i for some i, then it returns the evaluation of the method Plus which is t2, i.e. the instance of the object representing x,
- if t1 = suc(j) and t2 = i for some i and j, then it returns the evaluation of the method Plus which is the object representing the algebraic term suc whose argument is the sum of the subterm of t1 and t2.

Pattern matching enables the discrimination of algebraic constructors and deconstruction of algebraic terms into their subterms. It is an essential operation for the evaluation of terms through *term rewriting*. Term rewriting is a model of computation for replacing subterms of algebraic terms with other terms. It is often combined with equational theories as proposed in the work of Gerald Peterson and Mark Stickel [PS81] that was later generalized by Jean-Pierre Jouannaud and Hélène Kirchner [JK86]. Being a rewriting language, the Tom language provides pattern matching modulo associativity for those terms built from variadic operators. For instance, we define a variadic operator concNat for the algebraic type nat representing lists of Peano integers and calculate the sum of them :

```
nat = zero() | suc(n:nat) | concNat(nat*)
...
public nat ListPlus(nat t) {
  %match(t) {
    concNat()       -> { return `zero(); }
    concNat(x)      -> { return `x; }
    concNat(x,y,z*) -> { return(listPlus(concNat(Plus(`x,`y),z*)); }
  }
}
```

When considering lists as patterns, Tom applies pattern matching modulo associativity and neutral element to them by the use of star variables such as z* occurring in the arguments of a list. Indeed, z* represents a contiguous sublist of t whose head symbol is the same of the list in which it occurs, i.e. concNat. The variables x and y in their turn represent Peano integers whose head symbols are different from concNat. However, thanks to the theory of associativity and neutral element, the pattern concNat(x,y,z*) matches concNat(zero(),concNat(zero(),suc(zero()) indicating concNat(suc(zero())) as instance of z and zero() as instance of x and y.

In this thesis we take interest in the improvement of the expressiveness of the pattern matching provided by Tom by taking advantage of different levels of type hierarchy. Indeed, the evaluation of a matching-equation whose pattern is not a variable consists in verifying that the pattern and the term to be matched have the same head symbol. Nevertheless, the presence of subtyping relations between algebraic types additionally requires to ascertain that the term to be matched *reduces* to a term whose type is a subtype of the one of the pattern. A reduction sequence constitutes the evaluation of terms and expressions of a language. We describe Tom code evaluation through the definition of the big-step operational semantics of the Tom language according to the formalism defined by Gilles Kahn [Kah87]. The reduction relation on Tom terms and expressions is inspired by the one of the OCaml language presented by Didier Rémy [Ré02]. However, the evaluation of Tom pattern matching expressions considering subtyping is rather influenced by the simplification rules described by Delia Kesner [Kes91] for the compilation of pattern matching in order-sorted languages. The formalization of the Tom operational semantics is essential to show that our type system is safe. Accordingly, we prove that every Tom well-typed expression is either a Tom term or reduces to one (*progress* property) and also that every step of evaluation preserves types of any Tom expression (*preservation* or *subject reduction* property).

The Tom programming language is characterized by the support for analysis and transformation of various kind of tree-based documents. To help the programmer to specify and implement analysis and transformation tools, Tom provides other high level concepts and constructs in addition to algebraic signatures and pattern matching such as representation in canonical form, rewrite rules and strategies to control their application. The subject of this thesis is rooted in the study of typing algorithms for the rules and strategies of the Tom language. We particularly focus on supporting the definition of algebraic types featuring subtyping. We aim to propose an improved Tom language which

provides both stronger expressiveness and more safety able to ensure that the transformations described by the rewrite rules preserve the type of terms. The possibility to design safer compilers using Tom has a significant impact on avoiding the generation of ill-typed terms incompatible with their type signatures.

Contributions

This thesis can be seen as a three-stepped contribution to the Tom system, each new step building and improving upon the former, with the goal of getting both safer and more expressive. Needless to say, all theoretical developments were made with a strong practical aim in mind.

A type system for Tom

At the beginning of the thesis, Tom was able to perform some (very) small typing tasks : for example, it could infer the type of the variables in the conditions of the rewrite rules and propagate them to the occurrences (if any) of these variables in the right-hand side of the rules. However, without a definition of a formal type system, the phase of compilation concerning typing (i.e. both type checking and inference) were not well delimited. Consequently, the typing phase performed some tasks (as syntax desugaring) related to other phases which in their turn performed redundantly some typing tasks.

The first contribution of this thesis is thus to present a formalized type system for Tom algebraic terms, in order to obtain more safety over types. We then define a type checking system to verify that all *pure* algebraic terms of a Tom program are well-typed. By *pure* we mean algebraic terms not built from concrete data types. In addition, we define a constraint-based type inference system to infer types of variables occurring in (even non-pure) algebraic terms. This leads to a type inference algorithm able to simultaneously fill in type information of algebraic terms and type check them. This type checking is possible since both inferred types and type annotations are used in the type constraint generation and resolution.

The implementation of a typer using the rules defined for type inference naturally leads to various changes at developer level. The internal representation of the Tom code needs to be able to handle type constraints through the definition of specific data constructors for them. The side effect of this first implementation is a general code cleanup (refactorization, review of the code architecture, etc).

Both the formalization and implementation of the whole type system are presented in details in Chapter 3.

Subtyping

The next contribution of this thesis is to add subtyping features to the aforementioned type system in order to improve the expressiveness of the pattern matching constructs provided by Tom, while keeping the obtained safety. This step naturally builds upon the system from the first contribution and takes advantage of the different levels of class hierarchy provided by the Java host language.

The chosen solution is here to enrich the formal type system of Tom with subtyping as a partial order over types. This involves introducing a new rule into the formalized type checking system stating that if a type s_1 is a subtype of s_2 then every term of type s_1 is of type s_2 as well. Correspondingly, the introduction of subtyping into the constraint-based type inference system involves extending the constraint language with subtyping constraints in addition to the equality ones. In addition, the type inference problem reduces to solving a set of equations and inequalities.

The implementation of this new system with subtyping has consequences for users and developers. At user-level, it necessitates an additional construct for the declaration of subtyping relations : the **extends** construct is combined with **%typeterm** during the definition of type constructors. This in turn enables the user to specify the inheritance relation between algebraic types when defining a handwritten mapping. Thereby, the type hierarchy handling algebraic types is expected to respect the one handling their respective concrete types. Internally, this leads to the modification of the representation of the language to handle more type annotations but also type relations, as well as the definition of specific data constructors for subtyping constraints. This permits the implementation of a new algorithm for the simplification and resolution of subtyping constraints.

We formally describe the typing rules and constraint resolution algorithm as well as aspects of their implementation in Chapter 4.

Operational semantics

Finally, the integration of subtyping into Tom also impacts the matching algorithms implemented by the phase of compilation. Our third contribution is a redefinition of these matching algorithms.

In the presence of types partially ordered by subtyping, pattern matching consists of two kinds of verification :

- equality (even modulo an equational theory) of type constructors of pattern and subject corresponding to the structure matching as in non-ordered types ;
- ascertainment that the subject reduces to a term of type s_1 subtype of the type s_2 of the pattern.

Hence, the reduction rules composing the algorithms of syntactic matching, matching modulo $\mathcal{A}$ and matching modulo $\mathcal{AU}$ need also consider subtyping constraints. For this purpose, in Chapter 3 we define the Tom core syntax and

its operational semantics describing Tom code evaluation. Then, in Chapter 4 we adapt the reduction rules to allow for type inclusion at the pattern level. Since the Tom system does not analyze the host code, the code responsible for the type verification of the solutions of a match-equation must be translated into semantically equivalent instructions in the Java host language. In addition, the result of the integration of subtyping in Tom has been tested in a case study presented in Chapter 5 using Tom to handle model transformation.

Outline

This thesis is composed of five chapters. The first chapter introduces the definitions underlying rewriting-based languages. We then present the Tom rewriting language in the second chapter. The two succeeding chapters concern the type system proposed for Tom. A discussion about its usage and advantages is presented in the last chapter. All chapters are short summarized in the following :

Chapter 1 presents a few preliminary notions useful for a better understanding of the succeeding chapters : term algebra, equational theories and term rewriting.

Chapter 2 provides an informal presentation of the Tom language as the main support for this work. It explains the functionality and usage of the major Tom constructs to : define algebraic signatures representing concrete objects, define rewrite rules through syntactic and associative pattern matching over algebraic terms, simplify the definition of patterns, define elementary strategies, and apply strategies for controlling rewrite rules. It also presents the compilation process of Tom programs and a summary of the main contributions of this thesis in the scope of the development of Tom.

Chapter 3 introduces an important contribution of this thesis to the safety of the Tom language. It formalizes the Tom algebraic terms and the operational semantics of the Tom syntax. It also proposes an original notion of types and a type checking system with support for those algebraic terms built from variadic operators. Then, typing is shown to be correct with respect to semantics and preserved by evaluation. A constraint-based type inference system is another proposition of the chapter. The constraint resolution algorithm composed of rewrite rules is shown to be terminating and the type inference system is shown to be complete and sound with respect to the type checking system. Finally, the implementation and integration aspects of the inference type system are addressed.

Chapter 4 introduces the main purpose of this thesis which is the improvement of the expressiveness of the Tom language. It extends the operational semantics of Tom to allow for type inclusion at the level pattern. Under the assumptions of non-overloading and non-multiple inheritance, the preceding type system is enriched with the possibility of subtyping declara-

tions. This weakening of the notion of type compatibility is also considered in the subtyping-constraint-based type inference system proposed in the chapter. Accordingly, an algorithm for subtyping constraint solving is described as a combination of constraint propagation, generation of solution and garbage collecting. This algorithm is shown to be terminating and the precedent properties of safety, soundness and completeness are proved for the whole type system. The type inference system is then described in a practical viewpoint concerning data structures, algorithms and strategies required for its implementation.

Chapter 5 illustrates the advantages provided by the use of subtyping when programming in Tom. It promotes the subtyping-constraint-based type system whose typing rules cover a larger number of Tom constructs in comparison to the type system available in the beginning of this thesis. Some Tom programs are considered to benchmark the performance of the type system formally described in the two preceding chapters. Lastly, a case study of model transformations using Tom is presented as a real application of subtyping in Tom programming.

Chapter 1

Preliminary notions

In this chapter we introduce notations concerning term rewriting, equational theories and type systems. These concepts are detailed by Franz Baader and Tobis Nipkow [BN98] and also by Claude Kirchner and Hélène Kirchner [KK06]. We assume that the reader is familiar with the concepts of (multi)sets, relations and functions defined by Benjamin Pierce [Pie02] among others.

1.1 Terms

This section presents the basic notations on first-order term algebra. We will define term constructions using different paradigms.

Definition 1.1 (Signature)**.** *A* signature $\mathcal{F}$ *is a non-empty set of operators (function symbols), each one associated to a fixed* arity*.*

The arity *of an operator f is a natural number indicating the number of arguments it has. This arity is given by a function $ar : \mathcal{F} \to \mathbb{N}$. $\mathcal{F}_n$ is the subset of operators having n for arity, $\mathcal{F}_n = \{f \in \mathcal{F} \mid ar(f) = n\}$.*

The set $\mathcal{F}_0$ is called constant set *and its elements are called* constant symbols*.*

In order to classify terms we often consider terms divided into classes called *sorts*. In this case, we must define many-sorted signatures.

Definition 1.2 (Many-sorted signature)**.** *A* many-sorted signature *is a pair $(\mathcal{S}, \mathcal{F})$ where $\mathcal{S}$ is a set of sorts and $\mathcal{F}$ is a set of ranked function symbols $\mathcal{F}$.*

The rank of an operator $f \in \mathcal{F}$, denoted by $\mathcal{R}ank(f)$, is defined by the tuple $(s_1, \ldots, s_n, s)$ and is often denoted by $f : s_1, \ldots, s_n \to s$. This means that f has $ar(f) = n$ with domain $(s_1, \ldots, s_n)$, denoted by $dom(f)$, and codomain s denoted by $codom(f)$.

From a given signature we can define terms composed of operators and a countably infinite set $\mathcal{V}$ of variables. The set $\mathcal{V}_s$ denotes a set of variables of sort $s \in \mathcal{S}$. Definitions for usual (unsorted) terms are denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and can be easily obtained from the many-sorted case such that $\mathcal{S}$ is a singleton

$\{s\}$. In the rest of this chapter we focus on many-sorted terms. For this reason, hereinafter, we use term as a synonym for many-sorted term. We can define an algebraic structure over many-sorted signatures.

Definition 1.3 (Term algebra)**.** *$\mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V})$ is the term algebra built from a given nonempty set $\mathcal{S}$ of sorts, a finite set $\mathcal{F}$ of function symbols and a countably infinite set $\mathcal{V}$ of variables and is defined inductively:*

1. *$\mathcal{V}_s \subseteq \mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V})$: all variables of sort $s \in \mathcal{S}$ of $\mathcal{V}$ are terms of sort s of $\mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V})$,*
2. *$\forall t_i \in \mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V})$ of sort $s_i \in \mathcal{S}$, for $i \in [1,n]$, and $\forall f : s_1,\ldots,s_n \to s \in \mathcal{F}$, $f(t_1,\ldots,t_n) \in \mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V})$ and has sort s.*

When the set $\mathcal{S}$ is a singleton, the term algebra built from $\mathcal{S}$, $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$.

We denote the variables of $\mathcal{V}$ by either x, y, z or x_1, x_2, x_3, ... The head symbol of a term t with the form $f(t_1,\ldots,t_n)$ is denoted by $symb(t)$ and is by definition the operator f. Moreover, the terms t_1, ..., t_n are called *arguments* of f.

Definition 1.4 (Variables)**.** *The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$ and is defined inductively:*

- *$\mathcal{V}ar(t) = \emptyset$ if $t \in \mathcal{F}_0$,*
- *$\mathcal{V}ar(t) = \{t\}$ if $t \in \mathcal{V}$,*
- *$\mathcal{V}ar(t) = \bigcup_{i=1}^{n} \mathcal{V}ar(t_i)$ if $t = f(t_1,\ldots,t_n)$.*

Some classes of terms are of special interest in rewriting. *Linear* terms form such a class. A term $t \in \mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V})$ is said to be *linear* if each variable occurs at most once in t. Moreover, if no variable occurs in t, then t is called a *ground term* and $\mathcal{T}(\mathcal{S},\mathcal{F})$ is the set of ground terms.

The main difference between variables and constant symbols is that the former may be replaced by the application of substitutions.

Definition 1.5 (Substitution)**.** *Let $\mathcal{V}$ be a countably infinite set of variables and $\mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V})$ be a term algebra. A* substitution *σ is a total function from $\mathcal{V}$ to $\mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V})$.*

The set of variables such that $\sigma(x) \neq x$ is called the domain *of σ, denoted by $\mathcal{D}om(\sigma)$. The set of terms such that $\sigma(x) = t$ for all $x \in \mathcal{D}om(\sigma)$ is called the* range *of σ, denoted by $\mathcal{R}ange(\sigma)$. If $\mathcal{D}om(\sigma) = \{x_1,\ldots,x_n\}$, then we write σ as $\sigma = \{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$. This notion can be extended to an endomorphism $\sigma' : \mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V}) \to \mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{V})$ as follows:*

- *$\sigma'(x) = \begin{cases} \sigma(x) & \text{if } x \in \mathcal{D}om(\sigma) \\ x & \text{otherwise} \end{cases}$*
- *$\sigma'(f(t_1,\ldots,t_n)) = f(\sigma'(t_1),\ldots,\sigma'(t_n))$ where $f \in \mathcal{F}_n$.*

For the sake of readability, we will not distinguish between σ and σ'. A term t_1 is called an *instance* of a term t_2 if and only if there exists a substitution σ such that $\sigma(t_2) = t_1$.

Example 1.6. *The set of Peano integers can be built from* $\mathcal{S}^{pe} = \{\mathbb{N}\}$, $\mathcal{F}^{pe} = \{zero : \mathbb{N}, suc : \mathbb{N} \to \mathbb{N}, plus : \mathbb{N} \times \mathbb{N} \to \mathbb{N}\}$ *and* $\mathcal{V}^{pe} = \{x\}$. *Given the term* $t = plus(zero, suc(x))$ *and the substitution* $\sigma = \{x \mapsto zero\}$, *we have* $\sigma(t) = plus(zero, suc(zero))$.

Definition 1.7 (More general substitution)**.** *A substitution* σ_1 *is* more general *(or* less specific*) than a substitution* σ_2, *written* $\sigma_1 \lesssim \sigma_2$ *if there exists a substitution* δ *such that* $\sigma_2 = \delta\sigma_1$. *In this case we say that* σ_2 *is an instance of* σ_1.

Definition 1.8 (Unification)**.** *Let* $\mathcal{V}$ *be a countably infinite set of variables and* $\mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *be a term algebra. An* equation *is a couple* $(l, r) \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *denoted by an unquantified formula of the form* $l =^? r$. *An* unification problem *is a finite set of equations* $\mathcal{E}q = \{t_1 =^? t'_1, \ldots, t_n =^? t'_n\}$ *where* $t_i, t'_i \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$.

A solution *(or* unifier*) of an equation* $t_1 =^? t'_1$ *is a substitution* σ *such that* $\sigma(t_1) = \sigma(t'_1)$ *and we say that* σ unifies t_1 *and* t'_1.

A substitution is solution of $\mathcal{E}q$ *if it is solution of every equation in* $\mathcal{E}q$. $\mathcal{U}(\mathcal{E}q)$ *denotes the set of all solutions of* $\mathcal{E}q$. $\mathcal{E}q$ *is said* unifiable *if* $\mathcal{U}(\mathcal{E}q) \neq \emptyset$.

A substitution σ_1 is a *most general unifier (mgu)* of $\mathcal{E}q$ if σ_1 is a least element of $\mathcal{U}(\mathcal{E}q)$, i.e. for every solution $\sigma_2 \in \mathcal{U}(\mathcal{E}q)$, $\sigma_1 \lesssim \sigma_2$. The unification problem is also called *syntactic unification* and is decidable. The proof is e.g. done by Claude Kirchner and Hélène Kirchner (cf. [KK06]). Decidability of syntactic unification for computing most general unifiers has a fundamental importance for programming languages and many applications as type systems or other shown by Claude Kirchner [Kir90].

Subterms occurring in a term can be identified by their positions.

Definition 1.9 (Position)**.** *A position of a term* t *is represented by a finite sequence* ω *of non-zero natural numbers describing the path from the head position of* t *until a given subterm.*

The head position corresponds to the empty sequence and is denoted by ϵ.

The concatenation of two positions ω_1 *and* ω_2 *is denoted by* $\omega_1\omega_2$.

The subterm t *at the position* ω *is represented by* $t_{|\omega}$ *and is defined as follows:*

- $t_{|\epsilon} = t$,
- *if* $t = f(t_1, \ldots, t_n)$ *then* $\forall i \in [1, n], t_{|i\omega} = t_{i|\omega}$.

We denote $t[t']_\omega$ *the* replacement *of the subterm of* t *at position* ω *by* t'. *The set of positions of* t *is denoted by* $\mathcal{P}os(t)$.

Example 1.10. *Consider the terms of* $\mathcal{T}^{pe} = \mathcal{T}(\mathcal{S}^{pe}, \mathcal{F}^{pe}, \mathcal{V}^{pe})$. *The set of positions of the term* $t = plus(zero, suc(x))$ *of the term algebra* $\mathcal{T}^{pe}$ *is*

$$\mathcal{P}os(t) = \{\epsilon, 1, 2, 21\}$$

which corresponds respectively to the subterms $t_{|\epsilon} = plus(zero, suc(x))$, $t_{|1} = zero$, $t_{|2} = suc(x)$ *and* $t_{|21} = x$.

There is another way to inductively define terms: it uses a *inference system* format. This definition will be specially useful when talking about type systems in Chapter 3 and Chapter 4.

Definition 1.11 (Inference system)**.** *An* inference rule *is a rule with a label* $\mathcal{L}$ *which takes a (possibly empty) set of hypothesis (or premises)* $\mathcal{H}_1, \ldots, \mathcal{H}_n$ *and returns a conclusion* $\mathcal{C}$*:*

$$\frac{\mathcal{H}_1 \quad \ldots \quad \mathcal{H}_n}{\mathcal{C}}\ \mathcal{L}$$

An inference system *is a set of inference rules. In such a system, a conclusion* $\mathcal{C}$ *is said* derivable *in an inference system if and only if one of the following two conditions are satisfied:*

- *there exists a rule having no hypothesis and a conclusion* $\mathcal{C}$*,*
- *there exists a rule having* $\mathcal{C}$ *as conclusion and whose all hypothesis are derivable.*

The system is usually formulated as a set of rule schemas*, i.e. parametrized rules used to express a large family of rules by instantiation, as follows:*

$$\phi\ \frac{\mathcal{H}_1 \quad \ldots \quad \mathcal{H}_n}{\mathcal{C}}\ \mathcal{L}$$

where ϕ *is a proposition, called* application condition*, concerning hypothesis and conclusion. The application condition is used to limit the set of rules described by the schema and it disappears when instantiated.*

A derivation tree *of a conclusion is a sequence of derivation rules ensuring that such a conclusion is derivable.*

Finally, a special case of inductive definitions is the *algebraic grammar* used to formally define a language. It gives an explicit syntactic procedure for generating/recognizing the elements of a term algebra by concatenation of a set of symbols.

Definition 1.12 (Algebraic grammar)**.** *An algebraic grammar is a set of rules having the form* $\mathcal{N} ::= \ldots \mid (\mathcal{N} \cup \mathcal{M})^* \mid \ldots$ *where* $\mathcal{N}$ *is called the set of* nonterminal symbols *(which are placeholders for other symbols of the grammar),* $\mathcal{M}$ *is called the set of* terminal symbols *(which is disjoint from* $\mathcal{N}$*),* $\mathcal{N} \cup \mathcal{M}$ *means "a nonterminal or a terminal symbol" and* $\mathcal{X}^*$ *means "concatenation of zero or more symbols* $\mathcal{X}$*".*

The rules are composed of clauses *separated by the symbol "|". Each one of these rules is a notation for inference rules corresponding to as many clauses:*

- *the inference rules have conclusions "*$\mathcal{C} \in \mathcal{N}$*", where* $\mathcal{C}$ *is the clause associated to the current rule and the* $i-th$ *occurrence of a nonterminal symbol was replaced by* t_i*,*
- *the inference rules have premises "*$t_i \in \mathcal{N}_i$*" where* $\mathcal{N}_i$ *is the* $i-th$ *nonterminal symbol occurring in the clause associated to the rule.*

Example 1.13. *The following grammar specifies the language of the terms of* $\mathcal{T}^{pe}$*:*

$$\begin{array}{lcl} \mathcal{T}^{pe} & ::= & \mathcal{V}^{pe} \mid zero \mid suc(\mathcal{T}^{pe}) \mid plus(\mathcal{T}^{pe}, \mathcal{T}^{pe}) \\ \mathcal{V}^{pe} & ::= & x \end{array}$$

where "$\mathcal{T}^{pe}$*" and "*$\mathcal{V}^{pe}$*" are nonterminal symbols and "x", "a", "plus", ",", "(" and ")" are terminal symbols. By definition, these rules are a notation for the following inference rules:*

$$\frac{}{x \in \mathcal{V}^{pe}} \qquad \frac{t_1 \in \mathcal{V}^{pe}}{t_1 \in \mathcal{T}^{pe}} \qquad \frac{}{zero \in \mathcal{T}^{pe}} \qquad \frac{t_1 \in \mathcal{T}^{pe}}{suc(t_1) \in \mathcal{T}^{pe}}$$

$$\frac{t_1 \in \mathcal{T}^{pe} \qquad t_2 \in \mathcal{T}^{pe}}{plus(t_1, t_2) \in \mathcal{T}^{pe}}$$

The definition of notation and language's syntaxes can be done simultaneously:

$$\mathcal{T}^{pe} \ni t_1, t_2 ::= x \mid zero \mid suc(t_1) \mid plus(t_1, t_2)$$

where t_1 and t_2 denote elements of $\mathcal{T}^{pe}$.

The set of algebraic terms is often supplied with a binary relation on its elements, called an *order*.

Definition 1.14 (Order)**.** *A binary relation* $\preceq$ *on a set* $\mathcal{S}$*, denoted by* $(\mathcal{S}, \preceq)$*, is:*

- *a* preorder *on* $\mathcal{S}$ *if it is reflexive and transitive,*
- *a* partial order *on* $\mathcal{S}$ *if it is a preorder and it is also antisymmetric,*
- *a* total order *on* $\mathcal{S}$ *if it is a partial-order and any two elements of* $\mathcal{S}$ *are* comparable*, i.e. for each* $s_1, s_2 \in \mathcal{S}$*, either* $s_1 \preceq s_2$ *or* $s_2 \preceq s_1$*.*

A nonempty set $\mathcal{S}$ equipped with a partial order $\preceq$ is called a *partially ordered set*, or more briefly a *poset*. We write $s_1 \prec s_2$ to mean $s_1 \preceq s_2 \wedge s_1 \neq s_2$ and the relation $\prec$ is called a *strict order* on $\mathcal{S}$.

Definition 1.15 (Least upper and greatest lower bounds)**.** *Let* $(\mathcal{S}, \preceq)$ *be a poset and* $s_1, s_2 \in \mathcal{S}$*.*

- *An element* $lub \in \mathcal{S}$ *is said to be a* least upper bound *(or* join*) of* s_1 *and* s_2 *if* $s_1 \preceq lub$ *and* $s_2 \preceq lub$ *and for any element* $s_3 \in \mathcal{S}$ *with* $s_1 \preceq s_3$ *and* $s_2 \preceq s_3$*, we have* $lub \preceq s_3$*.*
- *An element* $glb \in \mathcal{S}$ *is said to be a* greatest lower bound *(or* meet*) of* s_1 *and* s_2 *if* $glb \preceq s_1$ *and* $glb \preceq s_2$ *and for any element* $s_3 \in \mathcal{S}$ *with* $s_3 \preceq s_1$ *and* $s_3 \preceq s_2$*, we have* $s_3 \preceq glb$*.*

The definition of *lub* and *glb* allows the introduction of an special kind of poset referred to as a *lattice*.

Definition 1.16 (Lattice)**.** *A poset* $(\mathcal{S}, \preceq)$ *is a* lattice *if for every* $s_1, s_2 \in \mathcal{S}$ *both* $lub(s_1, s_2)$ *and* $glb(s_1, s_2)$ *exist and are unique in* $(\mathcal{S}, \preceq)$*.*

Order-sorted terms require many-sorted signatures encoding information about sort relations.

Definition 1.17 (Order-sorted signature)**.** *An* order-sorted signature *is a 3-tuple* $(\mathcal{S}, \preceq, \mathcal{F})$ *where* $(\mathcal{S}, \mathcal{F})$ *is a many-sorted signature and* $(\mathcal{S}, \preceq)$ *is a partially ordered set.*

Definition 1.18 (Order-sorted term algebra)**.** *For an order-sorted signature* $(\mathcal{S}, \preceq, \mathcal{F})$*, an* order-sorted term algebra $\mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *is a term algebra such that:*
- *a variable* $x \in \mathcal{V}$ *of sort* $s \in \mathcal{S}$ *is a term of sort* s' *of* $\mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *if* $s \preceq s'$*,*
- $f(t_1, \ldots, t_n)$ *is a term of sort* s' *of* $\mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *if and only if* $f : s_1, \ldots, s_n \to s \in \mathcal{F}$ *such that* $s \preceq s'$ *and* $t_i \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *is of sort* $s_i \in \mathcal{S}$*, for* $i \in [1, n]$*.*

1.2 Equational theories

Terms are used to represent objects of computation and their semantics is defined by their associated theories.

Definition 1.19 (Equality)**.** *Let* $\mathcal{V}$ *be a countably infinite set of variables and* $\mathcal{T}$ *be a term algebra. An* equality *(or* equational axiom*) is a pair of two terms* l, r *such that* $l, r \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$*. Equalities are written as* $l = r$*. The variables of an equality are assumed to be universally quantified. The terms* l *and* r *are respectively called the* left-hand side *(lhs) and the* right-hand side *(rhs) of the equality* $l = r$*.*

An equational specification *is a pair* $(\mathcal{F}, \mathcal{E})$ *where* $\mathcal{E}$ *is a set of equalities* $l = r$*.*

Equalities can be used to define transformations on terms. This is done by the replacement of instances of the lhs with the corresponding instances of the rhs occurring in a term.

Definition 1.20 (Reduction relation)**.** *Let* $\mathcal{E}$ *be a set of equalities. The binary relation* $\to_{\mathcal{E}}$ *is a* reduction relation *on* $\mathcal{T}$ *and is defined by* $t \to_{\mathcal{E}} t'$ *if there exists:*
- *an equality* $l = r \in \mathcal{E}$*,*
- *a position* $\omega \in \mathcal{P}os(t)$*,*
- *a substitution* σ *such that* $t_{|\omega} = \sigma(l)$ *and* $t' = t[\sigma(r)]_\omega$*.*

Let us now recall some definitions over relations which are useful in order to describe an *equational theory.*

Definition 1.21 (Inverse relation)**.** *Let* $\to$ *be a binary relation on a set* $\mathcal{T}$ *whose elements are denoted by* t_1*,* t_2*,... We call* inverse relation of $\to$ *the relation* $\leftarrow$ *defined by the following inference system:*

$$\frac{t_1 \to t_2}{t_2 \leftarrow t_1}\ \textsc{Inv}$$

Definition 1.22 (Symmetric, transitive and reflexive transitive closure)**.** *Let* $\rightarrow$ *be a binary relation on a set* $\mathcal{T}$ *whose elements are denoted by* t_1, t_2,... *We call* symmetric closure of $\rightarrow$ *the relation* $\leftrightarrow$ *defined by the following inference system:*

$$\frac{t_1 \rightarrow t_2}{t_1 \leftrightarrow t_2}\ \textsc{Sym}_1 \qquad \frac{t_2 \rightarrow t_1}{t_1 \leftrightarrow t_2}\ \textsc{Sym}_2$$

We call transitive closure of $\rightarrow$ *the relation* $\overset{+}{\rightarrow}$ *defined by the following inference system:*

$$\frac{t_1 \rightarrow t_2}{t_1 \overset{+}{\rightarrow} t_2}\ \textsc{Trans}_1 \qquad \frac{t_1 \overset{+}{\rightarrow} t_2 \qquad t_2 \overset{+}{\rightarrow} t_3}{t_1 \overset{+}{\rightarrow} t_3}\ \textsc{Trans}_2$$

We call reflexive transitive closure of $\rightarrow$ *the relation* $\overset{*}{\rightarrow}$ *defined by the following inference system:*

$$\frac{}{t \overset{*}{\rightarrow} t}\ \textsc{Refl-Trans}_1 \qquad \frac{t_1 \overset{+}{\rightarrow} t_2}{t_1 \overset{*}{\rightarrow} t_2}\ \textsc{Refl-Trans}_2$$

We say that t_1 is *reducible* if there is a t_2 such that $t_1 \rightarrow t_2$; otherwise t_1 is *in normal form* (or *irreducible*). We call *normal form of* t_1 each irreducible element t_2 of $\mathcal{T}$ such that $t_1 \overset{*}{\rightarrow} t_2$.

Closures are also defined to the inverse relation. Composition of reduction relations $\rightarrow_1$ followed by $\rightarrow_2$ are denoted $\rightarrow_1 \circ \rightarrow_2$. We denote *transitive symmetric closure* and *reflexive transitive symmetric closure* of $\rightarrow$ by $\overset{+}{\leftrightarrow}$ and $\overset{*}{\leftrightarrow}$, respectively. By definition, $\overset{*}{\leftrightarrow}$ is an equivalence relation for all $\rightarrow$.

The reflexive transitive symmetric closure of $\rightarrow_{\mathcal{E}}$ is the *equational theory of* $\mathcal{E}$ denoted by $=_{\mathcal{E}}$. Equational theories built on $\mathcal{T}$ define equivalence classes of terms $t \in \mathcal{T}$. Among many well-known equational theories, we emphasize two theories which are useful in this thesis.

Definition 1.23 (Associativity and neutral elements)**.** *Let* $\mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *be a term algebra and* f *be an operator such that* $f \in \mathcal{F}_2$*. The* associativity *is the property defined by:*

$$\forall x, y, z \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V}), f(f(x, y), z) = f(x, f(y, z))$$

and f *is called* associative operator.

An operator $e \in \mathcal{F}$ *satisfying the property:*

$$\forall x \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V}) f(e, x) = x$$

is called a left neutral element *for* f*.*

An operator $e \in \mathcal{F}$ *satisfying the property:*

$$\forall x \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V}), f(x, e) = x$$

is called a right neutral element *for* f*.*

An operator $e \in \mathcal{F}$ *is called a* neutral element *for* f *if it is a both left and right neutral element for* f*.*

We denote by $\mathcal{A}$ and $\mathcal{AU}$ the equational theories built respectively from the associativity and the combination of associativity and a neutral element for the considered operator.

Example 1.24. *Consider the terms of* $\mathcal{T}(\mathcal{S}^{pe}, \mathcal{F}^{pe}, \mathcal{V}^{pe})$ *and let*

$$t = plus(plus(suc(zero), zero), zero)$$

The term

$$plus(suc(zero), plus(zero, zero))$$

is equivalent to t *modulo* $\mathcal{A}$ *when* $plus$ *is associative. In the same way, the terms* $plus(suc(zero), zero)$ *and* $suc(zero)$ *are equivalent to* t *modulo* $\mathcal{AU}$*, where* $zero$ *is the neutral element for* $plus$*.*

In this thesis, we often consider the definition of terms extended to *variadic operators* since they constitute the terms of the Tom programming language presented in Section 2.2.

Definition 1.25 (Many-sorted variadic signature)**.** *A* variadic operator v *is a function symbol whose arity is not fixed.*

A many-sorted variadic signature *is a couple* $(\mathcal{S}, \mathcal{F}^\star)$ *where* $\mathcal{S}$ *is a set of sorts and* $\mathcal{F}^\star$ *is a non-empty set of sorted variadic operators defined as* $\mathcal{F}^\star = \bigcup_{s_1^\star, s \in \mathcal{S}} \mathcal{F}^\star_{s_1^\star, s}$.

The rank of an operator $v \in \mathcal{F}^\star_{s_1^\star, s}$*, denoted by* $\mathcal{R}ank(v)$*, is defined by the tuple* $(s_1^\star, s)$ *and is often denoted by* $v : {s_1}^\star \to s$*. This means that* v *may have any arity with domain* $(s_1^\star)$*, denoted by* $dom(v)$*, and codomain* s *denoted by* $codom(v)$*. Thus, the possible arguments of* v *are all of sort* s_1*.*

The notions of order and term algebra can be extended to variadic operators by considering $\mathcal{F} \cup \mathcal{F}^\star$ as the finite set of function symbols.

Definition 1.26 (Term algebra, with variadic operators)**.** $\mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *is the term algebra built from a given nonempty set* $\mathcal{S}$ *of sorts, a finite set of function symbols defined by* $\mathcal{F} \cup \mathcal{F}^\star$ *and a countably infinite set* $\mathcal{V}$ *of variables and is defined inductively:*

1. $\mathcal{V}_s \subseteq \mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *: all variables of sort* $s \in \mathcal{S}$ *of* $\mathcal{V}$ *are terms of sort* s *of* $\mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$*,*
2. $\forall t_i \in \mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *of sort* $s_i \in \mathcal{S}$*, for* $i \in [1, n]$*, and* $\forall f : s_1, \ldots, s_n \to s \in \mathcal{F}$*,* $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *and has sort* s *(i.e.* $codom(f) = s$*),*
3. $\forall n \in \mathbb{N}^+, \forall t_i \in \mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *of sort* $s_1 \in \mathcal{S}$*, for* $i \in [1, n]$*, and* $\forall v : {s_1}^\star \to s \in \mathcal{F}^\star$*,* $v() \in \mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *and* $v(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *and are of sort* s*.*

Definition 1.27 (Order-sorted variadic signature)**.** *An* order-sorted variadic signature *is a 3-tuple* $(\mathcal{S}, \preceq, \mathcal{F}^\star)$ *where* $(\mathcal{S}, \mathcal{F}^\star)$ *is a many-sorted variadic signature and* $(\mathcal{S}, \preceq)$ *is a partially ordered set.*

Definition 1.28 (Order-sorted term algebra, with variadic operators)**.** *For an order-sorted signature* $(\mathcal{S}, \preceq, \mathcal{F} \cup \mathcal{F}^\star)$*, an* order-sorted term algebra with variadic operators $\mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *is a term algebra with variadic operators such that:*

- *a variable* $x \in \mathcal{V}$ *of sort* $s \in \mathcal{S}$ *is a term of sort* s' *of* $\mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *if* $s \preceq s'$*,*
- $f(t_1, \ldots, t_n)$ *is a term of sort* s' *of* $\mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *if and only if* $f : s_1, \ldots, s_n \to s \in \mathcal{F}$ *such that* $s \preceq s'$ *and* $t_i \in \mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *is of sort* $s_i \in \mathcal{S}$*, for* $i \in [1, n]$*,*
- $v()$ *and* $v(t_1, \ldots, t_n)$ *are terms of sort* s' *of* $\mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *if and only if* $v : {s_1}^\star \to s \in \mathcal{F}^\star$ *such that* $s \preceq s'$ *and* $t_i \in \mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *is of sort* $s_1 \in \mathcal{S}$*, for* $i \in [1, n]$ *and* $n \in \mathbb{N}^+$*.*

The associativity property and neutral elements are encoded in the structure of variadic terms by *flattening*.

Definition 1.29 (Flattening)**.** $\forall t_1, \ldots, t_n, t'_1 \ldots, t'_m \in \mathcal{T}(\mathcal{S}, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$,
$v(t_1, \ldots, t_{i-1}, v(t'_1, \ldots, t'_m), t_{i+1}, \ldots, t_n) \to_{\mathcal{AU}}$
$v(t_1, \ldots, t_{i-1}, t'_1, \ldots, t'_m, t_{i+1}, \ldots, t_n)$

where $n, m \in \mathbb{N}$*,* $v \in \mathcal{F}^\star$ *and* $v()$ *(i.e. the function* v *with no arguments) is defined as a neutral element for the variadic operator* v*.*

Example 1.30. *Consider the terms of* $\mathcal{T}(\mathcal{S}^{pe}, \mathcal{F}^{pe} \cup \mathcal{F}^{\star pe}, \mathcal{V}^{pe})$ *where* $\mathcal{F}^{\star pe} = \{concNat : \mathbb{N}^\star \to \mathbb{N}\}$ *and let* $t = concNat(zero, concNat(suc(zero)), concNat())$*. The terms* $concNat(zero, concNat(suc(zero)))$*,* $concNat(zero, suc(zero))$ *and* $concNat(suc(zero), zero)$ *are equivalent to* t *modulo* $\mathcal{AU}$*, where* $concNat()$ *is a neutral element for the variadic operator* $concNat$*.*

The notion of position and replacement can be extended to flattened variadic terms and defined in the same way as in the work of Claude Marché [Mar93]. Variadic terms are assigned to variables according to the definition of substitution described by Narges Berregeb, Adel Bouhoula and Michaël Rusinowitch [BBR96]. Thereby, the remaining definitions concerning term algebras can easily be extended to variadic terms.

1.3 Rewriting systems

Rewriting (as defined e.g. by Claude Kirchner and Hélène Kirchner [KK06]) is a theory of computing that models computations by stepwise transformations of objects. Its more abstract approach consists of abstract reduction systems independently of the nature of the objects that are rewritten. Among several kinds of abstract reduction systems, we are interested in the ones whose objects are algebraic terms: term rewriting systems.

Definition 1.31 (Abstract reduction system (ARS))**.** *An* abstract reduction system *(ARS) is a couple* $(\mathcal{T}, \to)$ *where* $\to$ *is a binary relation on the set* $\mathcal{T}$*, i.e.* $\to \subseteq \mathcal{T} \times \mathcal{T}$*.* $\to$ *is called a* rewrite relation *on* $\mathcal{T}$*.*

If we have $(t, t') \in \rightarrow$ for $t, t' \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$, then we write $t \rightarrow t'$ and say t *reduces* to t'. By $t \nrightarrow t'$ we denote that t does not reduce to t'. Moreover, a *reduction sequence* with respect to $\rightarrow$ is a sequence $(seq_n)_{n\in\mathbb{N}}$ with the form $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots$. A reduction sequence starting from t_0 is called a reduction sequence *of* t_0 and a *reduction step* is a specific occurrence of $\rightarrow$ in a reduction sequence. We define a number of properties of the rewrite relation $\rightarrow$.

Definition 1.32 (Weak and strong normalization)**.** *Let $(\mathcal{T}, \rightarrow)$ be an ARS:*

- *the relation $\rightarrow$ is* weakly normalizing *(or* weakly terminating*) if every element $t \in \mathcal{T}$ has a normal form,*
- *the relation $\rightarrow$ is* strongly normalizing *(or* terminating*) if every reduction sequence is finite.*

Definition 1.33 (Confluence and Church-Rosser property)**.** *Let $(\mathcal{T}, \rightarrow)$ be an ARS. The following two properties of $\rightarrow$ are equivalents:*

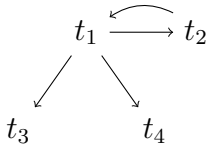
- *the relation $\rightarrow$ is* confluent *if for all t_1, t_2 and t_3 such that $t_1 \overset{*}{\rightarrow} t_2$ and $t_1 \overset{*}{\rightarrow} t_3$, there exists t such that $t_2 \overset{*}{\rightarrow} t$ and $t_3 \overset{*}{\rightarrow} t$,*
- *the relation $\rightarrow$ is* Church-Rosser *if for all t_2 and t_3 such that $t_2 \overset{*}{\leftrightarrow} t_3$, there exists t such that $t_2 \overset{*}{\rightarrow} t$ and $t_3 \overset{*}{\rightarrow} t$.*

These properties are illustrated by the following diagrams where lines and dotted lines denote a hypothesis and a conclusion, respectively.

The right and left diagrams represent respectively the confluence and the Church-Rosser property. A demonstration of the equivalence of both properties is e.g. shown by Claude Kirchner and Hélène Kirchner [KK06].

A relation $\rightarrow$ has the *unique normal form property* if it is (weakly or strongly) normalizing and confluent.

Example 1.34. *The relation illustrated in the diagrams below and defined by $t_1 \rightarrow t_2$, $t_2 \rightarrow t_1$, $t_1 \rightarrow t_3$ and $t_1 \rightarrow t_4$ is weakly normalizing but not strongly normalizing.*



It is not confluent and does not have the unique normal form property.

In order to get closer to computer programming, we present an ARS applied to algebraic terms.

Definition 1.35 (Term rewriting system (TRS))**.** *A* rewrite rule *for* $\mathcal{T} = \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *is an ordered pair* $(l, r) \in \mathcal{T} \times \mathcal{T}$ *such that* l *is not a variable and* $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$*. The terms* l *and* r *are respectively called the* left-hand side *(lhs) and the* right-hand side *(rhs) of the rule.*

A term rewrite system *(TRS)* $\mathcal{R}$ *for* $\mathcal{T}$ *is a set of rewrite rules for* $\mathcal{T}$*.*

A reduction relation $\rightarrow_{\mathcal{R}}$ *on* $\mathcal{T}$*, called* rewrite relation *on* $\mathcal{T}$*, is described similarly to Definition 1.20:* $(\mathcal{T}, \rightarrow_{\mathcal{R}})$ *where* $\rightarrow_{\mathcal{R}}$ *is a rewrite relation on* $\mathcal{T}$*.*

A term t rewrites *to a term* t'*, which is denoted by* $t \rightarrow_{\mathcal{R}} t'$*, if there exists:*

- *a rewrite rule* $(l, r) \in \mathcal{R}$*,*
- *a position* $\omega \in \mathcal{P}os(t)$*,*
- *a substitution* σ *such that* $t_{|\omega} = \sigma(l)$ *and* $t' = t[\sigma(r)]_\omega$*.*

The rewrite rules of $\mathcal{R}$ are denoted by $\rho_1, \rho_2, \rho_3, \ldots$. The rule (l, r), indicated by ρ, is denoted by $\rho : l \twoheadmapsto r$. A rewrite rule $\rho : l \twoheadmapsto r$ can be viewed as a schema. An instance of ρ is obtained by applying a substitution σ. The result is an *atomic* reduction step where the instance of the left-hand side $\sigma(l)$, called *redex*, rewrites to the instance of the right-hand side $\sigma(r)$, called *contractum.*

Example 1.36. *Consider again the terms of* $\mathcal{T}^{pe} = \mathcal{T}(\mathcal{S}^{pe}, \mathcal{F}^{pe}, \mathcal{V}^{pe})$*. The operator* $plus \in \mathcal{F}^{pe}$ *can be defined by the following rewrite rules:*

$$\mathcal{R}^{pe} = \left\{ \begin{array}{lcl} \rho_1^{pe} : plus(zero, x) & \twoheadmapsto & x \\ \rho_2^{pe} : plus(suc(y), z) & \twoheadmapsto & suc(plus(y, z)) \end{array} \right.$$

$\mathcal{R}^{pe}$ *is a TRS for* $\mathcal{T}^{pe}$ *and the term* $t = suc(plus(suc(zero), zero))$ *reduces to* $suc(suc(zero))$ *by the following reduction sequence:*

$$suc(plus(suc(zero), zero)) \rightarrow_{\mathcal{R}} suc(suc(plus(zero, zero))) \rightarrow_{\mathcal{R}} suc(suc(zero))$$

Moreover, the application of $\sigma = \{x \mapsto zero, y \mapsto zero, z \mapsto zero\}$ *to* $\mathcal{R}^{pe}$ *yields the atomic reduction steps:*

$$\begin{array}{lcl} \sigma(\rho_1^{pe}) : plus(zero, zero) & \twoheadmapsto & zero \\ \sigma(\rho_2^{pe}) : plus(suc(zero), zero) & \twoheadmapsto & suc(plus(zero, zero)) \end{array}$$

We simply write $\rightarrow$ instead of $\rightarrow_{\mathcal{R}}$ if $\mathcal{R}$ is obvious from the context. Moreover, often we specify a TRS only by its set of rewrite rules.

As said in Definition 1.35, an atomic reduction step $t \rightarrow t'$ can only happen when one has found a position $\omega \in \mathcal{P}os(t)$, a rewrite rule $(l, r) \in \mathcal{R}$ and a substitution σ such that $t_{|\omega} = \sigma(l)$. In this case, we say that l *matches* $t_{|\omega}$ and that σ is the *solution* of the *match-equation* $l \ll^? t_{|\omega}$.

Definition 1.37 (Matching)**.** *Let* p *be a pattern and* t *be a subject, where* $p \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *and* $t \in \mathcal{T}(\mathcal{S}, \mathcal{F})$*. A* match-equation *is an ordered pair* (p, t) *denoted by an unquantified formula of the form* $p \ll^? t$*.*

We say that p matches t*, or* t *is an instance of* p*, if there exists a substitution* σ *such that* $\sigma(p) = t$*. This is denoted by the formula* $p \ll t$*:*

$$p \ll t \Leftrightarrow \exists \sigma, \sigma(p) = t$$

and σ *is said* solution *of the match-equation* $p \ll^? t$*.*

A matching system *is a conjunction of match-equations.*

A substitution is solution of a matching system $\mathcal{P}$ *if it is solution of all the match-equations in* $\mathcal{P}$*.*

We write $p \not\ll t$ to mean that p does not match t.

Example 1.38. *Continuing Example 1.36 with TRS* $\mathcal{R}^{pe}$*, the substitution* $\sigma = \{x \mapsto zero, y \mapsto zero, z \mapsto zero\}$ *and the terms* $t = suc(plus(suc(zero), zero))$ *and* $t' = suc(suc(plus(zero, zero)))$ *and* $t'' = suc(suc(zero))$*. Accordingly,* $\sigma(lhs(\rho_2^{pe})) \ll t_{|1}$ *and* $\sigma(lhs(\rho_1^{pe})) \ll t'_{|11}$*. Thus, by* $\mathcal{R}^{pe}$*:*

$$\underbrace{suc(plus(suc(zero), zero))}_{t} \rightarrow \underbrace{suc(suc(plus(zero, zero)))}_{t'=t[\sigma(rhs(\rho_2^{pe}))]_1} \rightarrow \underbrace{suc(suc(zero))}_{t''=t'[\sigma(rhs(\rho_1^{pe}))]_{11}}$$

as seen in Example 1.36.

The matching problem is also called *syntactic matching* and is decidable. The proof is e.g. done by Claude Kirchner and Hélène Kirchner (cf. [KK06]).

1.3.1 Rewriting modulo an equational theory

Rewrite rules in a TRS have the same importance as the equalities in an equational specification. Given an equational theory $\mathcal{E}$, the equalities of $\mathcal{E}$ can be oriented into rewrite rules in order to solve the *word problem*: deciding whether two terms are equals modulo $\mathcal{E}$. However, the better orientation for those rewrite rules are not always clear and, in many cases, there is no way of orienting the equalities without loosing the normalization or confluence properties. A typical example is the equational theory of commutativity, built from the equality $\forall x, y \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V}), f(x, y) = f(y, x)$ whose transformation into a rewrite rule results in non-normalization. In that case, equational reasoning requires a different rewrite relation such as the one proposed by Gerald Peterson and Mark Stickel [PS81] and generalized by Jean-Pierre Jouannaud and Hélène Kirchner [JK86], called *rewriting modulo an equational theory.*

Definition 1.39 (Rewriting modulo an equational theory)**.** *Let* $\mathcal{R}$ *be a rewrite system and* $\mathcal{E}$ *be an equational theory. The term* t *rewrites to* t' *by* $\mathcal{R}$ *modulo* $\mathcal{E}$*, which is denoted by* $t \rightarrow_{\mathcal{R},\mathcal{E}} t'$*, if there exists:*

- *a rewrite rule* $(l, r) \in \mathcal{R}$*,*
- *a position* $\omega \in \mathcal{P}os(t)$*,*
- *a substitution* σ *such that* $t_{|\omega} =_{\mathcal{E}} \sigma(l)$ *and* $t' =_{\mathcal{E}} t[\sigma(r)]_\omega$*.*

Example 1.40. *Consider the TRS* $\mathcal{R}^{pe}$ *and the term*

$$t = plus(plus(zero, suc(zero)), zero)$$

There exists only one reduction sequence of t*:*

$$seq_1 : t \to_{\mathcal{R}} plus(suc(zero), zero) \to_{\mathcal{R}} suc(plus(zero, zero)) \to_{\mathcal{R}} suc(zero)$$

Now, consider the equational theory $\mathcal{A}$ *of associativity. Then there exists two extra reduction sequences of* t *modulo* $\mathcal{A}$*, since* $t =_{\mathcal{A}} plus(zero, plus(suc(zero), zero))$*:*

$$\begin{aligned} seq_2 : t &\to_{\mathcal{R},\mathcal{A}} plus(zero, suc(plus(zero, zero))) \to_{\mathcal{R},\mathcal{A}} plus(zero, suc(zero)) \\ &\to_{\mathcal{R},\mathcal{A}} suc(zero) \\ seq_3 : t &\to_{\mathcal{R},\mathcal{A}} plus(zero, suc(plus(zero, zero))) \to_{\mathcal{R},\mathcal{A}} suc(plus(zero, zero)) \\ &\to_{\mathcal{R},\mathcal{A}} suc(zero) \end{aligned}$$

In the relation $\to_{\mathcal{R},\mathcal{E}}$, the notion of matching is replaced by that of *matching modulo an equational theory* as defined in the following.

Definition 1.41 (Matching modulo an equational theory)**.** *Let* $\mathcal{E}$ *be an equational theory,* p *be a pattern and* t *be a subject, where* $p \in \mathcal{T}(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *and* $t \in \mathcal{T}(\mathcal{S}, \mathcal{F})$*. We say that* p $\mathcal{E}$-matches t *(i.e.* p *matches* t *modulo* $\mathcal{E}$*), if there exists a substitution* σ *such that* $\sigma(p) =_{\mathcal{E}} t$*. This is denoted by the formula* $p \ll_{\mathcal{E}} t$*:*

$$p \ll_{\mathcal{E}} t \Leftrightarrow \exists \sigma, \sigma(p) =_{\mathcal{E}} t$$

and σ *is said* $\mathcal{E}$-solution *of the match-equation* $p \ll_{\mathcal{E}}^{?} t$*.*

Example 1.42. *Consider the equational theory* $\mathcal{A}$ *and the terms* $p, t \in \mathcal{T}(\mathcal{S}^{pe}, \mathcal{F}^{pe}, \mathcal{V}^{pe})$ *such that* $p = plus(x, y)$ *and* $t = plus(plus(zero, suc(zero)), zero)$*. The matching equation* $p \ll_{\mathcal{E}}^{?} t$ *has two* $\mathcal{E}$*-solutions:*

$$\sigma_1 = \{x \mapsto plus(zero, suc(zero)), y \mapsto zero\}$$

and

$$\sigma_2 = \{x \mapsto zero, y \mapsto plus(suc(zero), zero)\}$$

1.3.2 Conditional rewriting

The application of rewrite rules of a TRS can be controlled by logical conditions. In this sense, conditional rewriting is considered as a computational paradigm which integrates functional and logic programming.

Definition 1.43 (Conditional rewriting system (CTRS))**.** *A* conditional rewriting system *(CTRS)* $\mathcal{R}$ *for* $\mathcal{T}$ *consists of rules*

$$\rho : l \to r \ \textit{if}\ c$$

where c *is a conjunction of conditions* $t_i =^{?} u_i$ *such that* $l, r, t_i, u_i \in \mathcal{T}$*. For all rules in* $\mathcal{R}$*,* $\mathcal{V}ar(r) \cup \mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$*.*

We denote ρ_u as the (usual) unconditional rewrite rule obtained by stripping ρ of its conditions. The TRS that results from a CTRS $\mathcal{R}$ by considering all rewrite rules ρ_u is denoted by $\mathcal{R}_u$. Furthermore, different rewrite relations are obtained depending on how the operator $=^?$ is interpreted. Three natural ones were proposed by Nachum Dershowitz, Mitsuhiro Okada and G. Sivakumar [DOS88]:

1. *semi-equational systems* with conditions $t_i = u_i$,
2. *join systems* with conditions $t_i \downarrow u_i$, where $t_i \xrightarrow{*} w$ and $u_i \xrightarrow{*} w$ for some term $w \in \mathcal{T}$,
3. *normal systems* with conditions $t_i \xrightarrow{*} u_i$, where u_i is in normal form

Once we have presented definitions about term algebra, computations in the term algebra, term rewriting systems and their extensions, we are ready to see the application of these notions in practice. For this purpose, one of the aims of Chapter 2 is to present the Tom language which extends some well-known imperative programming languages by the addition of such concepts.

Chapter 2

The Tom language in the context of term rewriting

This thesis takes place in the context of writing error-free programs in the Tom rewriting language detailed by Emilie Balland et al [BBK+07]. Accordingly, we focus on the conception of a type system for this language as a formal method for specification and proof of desired behaviors of a Tom program. The Tom language applies the concepts related to terms and term rewriting defined in Chapter 1. An informal presentation of the Tom language is provided in the present chapter but a more complete presentation as well as its main ideas can be found respectively in the Tom manual reference [BBB+11] and in the work of Pierre-Etienne Moreau, Christophe Ringeissen and Marian Vittek [MRV03]. Furthermore, we present, in the end of this chapter, the improvements provided by this thesis for the Tom system.

2.1 Motivation

In computer programming, formal methods have been developed for helping guarantee that the behavior of a system corresponds exactly to its programmer's expectation. The desired behavior is described by a specification which must be proved (by *theorem proving*), in a mathematical way, in order to ensure that the system behaves correctly with respect to it. Specifications require to be (at least partially) manually written and demand a good knowledge on the adopted logic in order to guide and control the process of proving. For that reason, other well-known and less powerful formal approaches are often considered to check (by *model checking*) whether a finite model of the system attends the requirements of the specification or to monitor (by *runtime monitoring*) whether the execution of the system behaves according to the specification. Nevertheless, another modest and well-known formal methods are *type systems* which aim to prove the absence, not of unexpected computed results, but of meaningless terms (or instructions) in regard to a type specification.

Type systems are applied to programming languages with the purpose of

classifying terms according to the kinds (i.e. sorts) of values they compute when executed. By the typing mechanism, the runtime behaviors of the terms in a program can be calculated at compile time considering its type specification, i.e. the sorts assigned to terms.

When extending a given programming language with new dedicated features, the typing of these features must be compatible with the ones in the original language. The Tom language is a typical example of such a situation as it consists of an extension of Java [2] which reifies the concept of term rewriting. In this thesis, we are interested in defining and implementing a type system within the context of the development of the Tom rewriting language. Moreover, in order to enhance the expressiveness of our type system, we introduce subtyping as a partial order over sorts which are interpreted as unsorted terms.

2.2 The Tom language

The Tom language has been developed in the Pareo team-project [3] since 2001. Being a domain-specific language (DSL) embedded in Java, Tom provides pattern matching and rule based constructs allowing rewriting systems to be encoded in object-oriented source code. This kind of extension helps the adoption of the Tom language by non-specialized programmers. Moreover, Tom programs are compiled into Java enhancing both the durability of the programs and their integration in a large diversity of architectures. Indeed, the Tom language is used in many academic and industrial projects. For the first case, we mention:

- rewrite systems such as in the implementation of a completion mechanism defined by Guillaume Burel in his thesis [Bur09] for a particular tableau method named TaMeD [Bon04],
- prototype of languages with specific semantics such as the MiniML language whose compilation into Tom is described in Paul Brauner's thesis [Bra10],
- proof assistants such as the Lemuridae assistant developed by Paul Brauner for a particular sequent calculus named LKMS [Bra10],
- compilers such as the Tom compiler that is written in Tom and bootstrapped,
- model transformations such as those conceived for real-time embedded systems in the scope of the Quarteft [4] project.

As a recent industrial project we emphasize the Crystal Reports software developed by the Business Object Company. In this project, Tom is used to

2. The languages C, C++, C# and Python are also supported by Tom in a more modest scale.

3. Reference available at http://pareo.loria.fr . Visited on February 2012.

4. The Quarteft project (available at http://quarteft.loria.fr and visited on February 2012) associates the laboratories LAAS-CNRS (Laboratoire d'Analyse et d'Architecture des Systèmes), IRIT (Institut de Recherche en Informatique de Toulouse), ONERA-DTIM (The French Aerospace Lab) and Inria Nancy (the Pareo team) and the companies AIRBUS and Ellidiss

transform online analytical processing (OLAP) queries in queries of the Structured Query Language (SQL) in order to preserve the compatibility between older and newer versions of the software.

2.2.1 Algebraic signatures and terms

Although being an embedded language, the pattern matching constructs provided by Tom can be applied over algebraic terms independently of their concrete representation. In this sense one must specify the algebraic (and abstract) data types of the terms handled by Tom, the concrete data types used to represent them in the host language as well as a *mapping* defining how this representation is done. According to this mapping, the compiler transforms the applications of pattern matching constructs into manipulations of the concrete data types. In this section we present some extracts of the Tom code found in Appendix A to illustrate the use of the Tom constructs.

Mappings

A mapping is composed of the declarations of the algebraic data types and their respective implementations. Type constructors are defined by the Tom construct **%typeterm** while data constructors are defined by the Tom constructs **%op** and **%oplist**. To a better understanding of the use of these constructors, we consider the following Java classes JNat and JNatList which implement algebraic types:

```
1  static class JNat { }
2  static class JNatList { }
```

And we also consider the Java classes Jzero, jSuc and JconcJNat implementing algebraic operators:

```
1  static class Jzero extends JNat {
2    public Jzero() {}
3    public boolean equals(Object o) {
4      if(o instanceof Jzero) {
5        return true;
6      }
7      return false;
8    }
9  }

11 static class Jsuc extends JNat {
12   public JNat n;
13   public Jsuc() { }
14   public Jsuc(JNat n) { this.n = n; }
15 }

17 static class JconcJNat extends JNatList {
18   public JNat head;
```

```
19    public JNatList  tail ;
20    public JconcJNat() { head = null;  tail  = null; }
21    public JconcJNat(JNat h, JNatList  ntail ) {
22      head = h;
23       tail  = ntail ;
24    }
25    public boolean isEmpty() {
26      return (head == null && tail == null);
27    }
28  }
```

Let us now define a mapping between an algebraic data type and its implementation by these Java classes. We start by defining type constructors for the algebraic types Nat and NatList:

```
1   %typeterm Nat {
2     implement    { JNat }
3     is_sort(s)    { (s instanceof JNat) }
4     equals(t1,t2) { (t1.equals(t2)) }
5   }

7   %typeterm NatList {
8     implement    { JNatList }
9     is_sort(s)    { (s instanceof JNatList) }
10    equals(t1,t2) { (t1.equals(t2)) }
11  }
```

The algebraic types Nat and NatList are respectively implemented by the concrete types JNat and JNatList. Algebraic types are known only by Tom, since they are not declared in the resulting Java program. By the **%typeterm** construct, Tom know the concrete type of objects implementing algebraic terms of a given algebraic type. Moreover, for each algebraic type we can define the **is_sort** construct to allow checking the concrete type of an object implementing an algebraic term. Another useful construct is the **equals** which specify how to check the equality between two objects represented by algebraic terms of the current type. The ranked operators of a given algebraic type are defined by the **%op** construct which allows specifying how to construct and to destruct algebraic terms with such operators. We define the operators zero and suc for the type Nat as in the following:

```
1   %op Nat zero() {
2     is_fsym(s) { (s instanceof Jzero) }
3     make()     { new Jzero() }
4   }

6   %op Nat suc(n:Nat) {
7     is_fsym(s)    { (s instanceof Jsuc) }
8     get_slot(n,s) { ((Jsuc)s).n }
9     make(t0)      { new Jsuc(t0) }
10  }
```

The first line of the **%op** construct defines the many-sorted signature of the algebraic operator and, in case of a non-constant operator, the name of the subterms occurring in a term built from it. The **make** construct specifies how to build an object represented by an algebraic term whose head symbol is the current operator and subterms are those given as parameters. Furthermore, all objects represented by algebraic terms built from a given operator can be destructed (i.e. decomposed) by the **is_fsym** and **get_slot** constructs. The first construct allows checking if the head symbol of the object is the current operator. The last construct allows selecting subterms of the object by their names.

Operator overloading is not allowed in Tom and therefore different ranks can not be attributed to a same operator. Similarly to the syntactic ones, variadic operators of an algebraic type are defined by the **%oplist** construct where algebraic terms built from variadic operators are often called *lists*. When defining a list, the user must specify how to construct and destruct a list, i.e. how to build an empty list and insert an element in a list and how to get both the first element and the rest of a list. We define the list concNat for the type NatList as in the following:

```
1  %oplist NatList concNat(Nat*) {
2    is_fsym(s)          { (s instanceof JconcJNat) }
3    get_head(l)         { ((JconcJNat)l).head }
4    get_tail(l)         { ((JconcJNat)l). tail  }
5    is_empty(l)         { ((JconcJNat)l).isEmpty() }
6    make_empty()        { new JconcJNat() }
7    make_insert(t,l)    { new JconcJNat(t,l) }
8  }
```

The first line of the **%oplist** construct defines the many-sorted variadic signature of the list which has an indefinite arity with domain ($\mathsf{Nat}^\star$) as explained in Definition 1.25. The **is_fsym** construct is defined as previously. The **get_head** and **get_tail** constructs allow decomposing an object implementing a list in its first element and in a sublist containing its remaining elements. The **is_empty** construct allows checking if the object implements an empty list. In addition, the **make_empty** construct specify how to build an object represented by an empty list and the **make_insert** construct allow inserting an element at the first position of the object implementing a list.

Gom algebraic signatures

The definition of a mapping between Tom algebraic data types and Java concrete data types enables Tom to manipulate Java terms as if they were Tom terms. For a standard utilization of Tom, these mappings can be automatically generated by the Gom tool written by Antoine Reilles and thoroughly presented in his work [Rei07]. From a many-sorted algebraic signature, Gom generates both a mapping for the algebraic data types described by the signature and a set of Java classes implementing it.

Example 2.1. *The handwritten mapping presented heretofore can be generated by Gom from the following algebraic signature:*

```
1   module ExampleGom
2   abstract syntax

4   Nat = zero()
5       | suc(n:Nat)

7   NatList = concNat(Nat*)
```

The first line presents the name of the module and enables to encapsulate all generated classes in a Java package named examplegom. Other modules can be imported by indicating their names. The second line denotes the beginning of the definition of type constructors and their respective data constructors.

The Java classes generated by Gom are more complex than the handwritten ones. The resulting implementation uses maximal sharing and preserves the types of the algebraic signature, i.e. the concrete type of objects is conform to the algebraic type of the corresponding algebraic terms. Besides, Gom provides hooks that allow defining properties of the algebraic terms by specifying a rewrite system. In particular, normal forms for the algebraic terms by the application of an equational theory as $\mathcal{A}$, $\mathcal{AU}$ or either $\mathcal{ACU}$ (the associative-commutative with neutral element theory).

The backquote construct

Algebraic terms are built from a Tom mapping by the use of the ‘ (backquote) construct. This construct is used in the Java code in order to enable the construction of an object representing an algebraic term. This means that algebraic terms only known by Tom can be manipulated in Java programs too. Thus, if we consider the handwritten mapping presented previously, we can build the object one of the concrete type JNat representing the algebraic term suc(zero()) of the algebraic type Nat:

```
JNat one = ‘suc(zero());
```

We usually name corresponding concrete and algebraic types identically to simplify and make more intuitive the use of algebraic terms. This is also automatically done by Gom when generating mappings. For instance, if we consider the Gom algebraic signature defined earlier, the same object one is build as previously but having type Nat:

```
Nat one = ‘suc(zero());
```

The ‘ construct enables the user to manipulate algebraic terms without knowledge of their concrete implementations. Moreover, Java functions and Java objects manipulating Tom terms (i.e. algebraic terms) can also be seen as algebraic operators as follows:

```
Nat two = `suc(one);
```

In this line, the Java operator one is used in the construction of the algebraic term suc(suc(zero())) represented by the Java operator two. However, since Tom does not analyze Java code, one assumes that the Java expressions occurring in algebraic terms respect the signatures of the respective algebraic operators.

2.2.2 Matching in Java

The notion of algebraic types and terms are often not present in the majority of imperative languages such as Java. Moreover, these languages do not provide matching constructs which constitute the essentials of functional and rewriting-based languages.

The Tom language allows languages such as Java to be extended with the **%match** construct which implements the notion of pattern matching. Therefore, Tom programs can check whether a given pattern occurs in data structures of the host language and instantiate variables according to the solution of the match-equation.

The semantics of the **%match** construct is close to the *match* that exists in functional programming languages, but in an imperative context. A **%match** is parameterized by a list of subjects (i.e. expressions evaluated to ground terms) and contains a list of rules. The sorts of patterns which are expected to match each one of these subjects can *optionally* be declared. The left-hand side of the rules are patterns built upon constructors and fresh variables, without any linearity restriction. The right-hand side is *not* a term, but a Java statement that is executed when the pattern matches the subject. However, thanks to the ` construct a term can be easily built and returned. In a similar way to the standard **switch/case** construct, patterns are evaluated from top to bottom. In contrast to the functional *match*, several actions (i.e. right-hand sides) may be fired for a given subject as long as no control flow statement to interrupt execution (as **return** or **break**) is found.

Example 2.2. *Consider the handwritten mapping presented in Section 2.2.1. The addition of Peano integers may be encoded as follows:*

```
1  public JNat peanoPlus(JNat t1, JNat t2) {
2    %match(Nat t1, Nat t2) {
3      zero(), x -> { return `x; }
4      suc(y), x -> { return `suc(peanoPlus(y,x)); }
5    }
6    return null;
7  }
```

The Java method peanoPlus takes two terms t1 and t2 of type JNat representing the Peano integers and return the sum of them which also has type JNat. This calculation is done by matching (as in Definition 1.37):

- in the first rule, zero() $\ll$ t1 and x $\ll$ t2. Then the return statement returns the evaluation of the method peanoPlus which is t2, i.e. the instance of the object representing x.
- in the second rule, suc(y) $\ll$ t1 and x $\ll$ t2. This means that the head symbol of t1 is suc and that the subterm $t1_{|1}$ is an instance of y. Then the return statement returns the evaluation of the method peanoPlus which is the object representing the algebraic term suc whose argument is the sum of $t1_{|1}$ and t2.

The example uses the ‘ construct to build algebraic terms, enabling the manipulation of variables x and y into the Java instruction. Furthermore, this construct also allows the Java method peanoPlus to be both seen as an algebraic operator and recursively called, in the right-hand side of the second rule.

The two variables x occurring in the patterns have the same name but are not the same, since they appear in different rules and are then considered modulo renaming. Moreover, a special notation _ refers to the *wild card* which can be used to represent unnamed variables when they are not referred in the rest of the rule. In this case, two (or more) occurrences of _ intend to different anonymous variables. However, the use of non-linear terms as patterns is also possible in Tom where the instance of variables of the same name must be equal.

We note that the **%match** constructs are clearly integrated in the host language keeping the readability of the code. Moreover, the right-hand side of the rules contains Java code that can contain Tom code and so forth. The Tom compiler does not analyze the host code and consider it as a simple text including matching constructs. Only these constructs are translated into instructions of the host language. For instance, the method peanoPlus defined in Example 2.2 is translated by the Tom compiler into the following Java code:

```
1  public JNat peanoPlus(JNat t1, JNat t2) {
2    if (t1 instanceof JNat) {
3      if (t1 instanceof Jzero) {
4        if (t2 instanceof JNat) { return ((JNat ) t2); }
5      }
6    }

8    if (t1 instanceof JNat) {
9      if (t1 instanceof Jsuc) {
10       if (t2 instanceof JNat) {
11         return new Jsuc(peanoPlus(((Jsuc)t1).n,(( JNat ) t2 )));

13       }
14     }
15   }
16   return null;
17 }
```

The **%match** construct is translated into nested **if** (...) {...} **else** {...} and the algebraic terms and types are replaced by their corresponding definition given by the mapping. For instance, in line 3, the head symbol of the object t1 is checked

by the instruction (t1 **instanceof** JZero) which defines the **is_fsym** construct of the algebraic operator zero.

Associative matching over variadic operators

For those patterns composed of lists (i.e. variadic operators), the Tom language provides pattern matching modulo $\mathcal{AU}$[5]. The matching modulo $\mathcal{AU}$ is called *list-matching* and constitutes an instance of the matching modulo an equational theory introduced by Definition 1.41.

When considering lists as patterns, the notion of list-matching can be applied to them by the use of variables annotated with *. These variables are called *star variables* and occur in the arguments of lists. Star variables differ from the simple ones because they represent a contiguous sublist of the subject. This stands for a star variable representing a sublist whose head symbol is the same of the list in which it occurs. Besides, a neutral element for a given variadic operator is the empty list built from it. Therefore, considering the variadic operator concNat defined by a handwritten mapping in Section 2.2.1 and the match-equation

$$\mathsf{concNat(x*,y*)} \ll^{?} \mathsf{concNat(zero(),suc(zero(),suc(suc(zero()))))}$$

a possible interpretation for concNat is the concatenation operator :: for either Peano integers and lists of Peano integers. Thereby Peano integers are associated to the data constructors of type Nat. This stands for concNat(x*,y*) representing $(x :: y)$ and concNat(zero(),suc(zero(),suc(suc(zero())))) representing $(0 :: 1 :: 2)$. In addition, the empty list concNat() represents a neutral element $()$ for the concatenation operator. The solutions proposed by Tom for the list-matching problem are the following:

x	y
$()$	$(0 :: 1 :: 2)$
(0)	$(1 :: 2)$
$(0 :: 1)$	(2)
$(0 :: 1 :: 2)$	$()$

We note that the variables x* and y* correspond to lists represented by Java objects of type JNatList. As the simple ones, star variables also can be unnamed and denoted by _*. Furthermore, since a list-matching problem can have more than one solution, Tom enumerates all solutions disorderly except if a control flow statement in the action which the matching.

5. The matching modulo associativity and commutativity is also provided by the current Tom version 2.9.

Matching conditions

As in business rule management systems, the Tom language also allows more expressive left-hand side of rules to be encoded. Thanks to the extensions proposed by Radu Kopetz in his thesis [Kop08], the application condition (i.e. the left-hand side) of a rule can be composed of a set of matching and anti-matching conditions. The notion of anti-matching is formally defined in the work of Radu Kopetz et al. [KKM07, KKM08, CKKM10].

For the sake of expressiveness, the Tom language provides an additional syntax of the **%match** construct which becomes a list of pairs (condition,action). Each condition corresponds to one or more conditions combined by the boolean connectors && and || representing respectively the conjunction and the disjunction operators. Matching conditions assume the form pattern << subject (or the form pattern << patternType subject allowing types to be explicit) expressing affirmative conditions. Negative conditions can be stated by the notion of complement of patterns introduced by the symbol !. The complement symbol ! can occur in any position of the pattern and as many times as necessary constituting, hence, an anti-pattern. Additionally, numeric conditions are also supported by Tom enabling the comparison of terms by the use of the operators < (is less than), <= (is at most), > (is greater than), >= (is at least), == (is equal to) or != (is not equal to).

Example 2.3. *Consider the handwritten mapping presented in Section 2.2.1. Given two Peano integers* t1 *and* t2*, we can check if* t1 *is less or equal to* t2 *by the following Java method:*

```
1  public boolean leq(JNat t1, JNat t2) {
2    %match {
3      (t1 == t2)   -> { return true; }
4      zero() << t1 -> { return true; }
5      zero() << t2 -> { return false; }
6      suc(x1) << t1 && suc(x2) << t2 -> { return `leq(x1,x2); }
7    }
8    throw new RuntimeException("ErrorFound in leq.");
9  }
```

The Java method leq takes two terms t1 and t2 of type JNat representing the Peano integers and compare them: if the integer represented by t1 is less or equal to that one represented by t2, then it returns **true**; otherwise, it returns **false**. The first rule applies a numeric condition to compare t1 and t2. The other rules use conjunction of matching conditions. In the action of each rule, the Java statement **return** interrupts the execution flow since other treatments are no longer necessary.

2.2.3 Useful extensions

More constructs are provided by extensions of the Tom core language in order to simplify the definition of patterns. Such extensions are not essentials

but improve the concision and readability of programs written in Tom. At present we focus on those extensions used in the rest of this thesis. Interested readers are referred to the Tom manual reference [BBB+11].

Alias for patterns

Subterms of patterns can be attributed to an alias by the use of the @ construct. While solving a matching (or anti-matching) condition, the variable representing an alias is instantiated by the instance of the corresponding subterm of the subject. This variable can be used as any other variable in both condition and action.

Example 2.4. *Consider again the handwritten mapping presented in Section 2.2.1. Given a list of Peano integers, the number of arguments representing positive integers can be printed by the following Java method:*

```
1  public void printPositiveIntegers (JNatList nList) {
2    int counter = 0;
3    %match {
4      concNat(x*,pnat@suc(y),z*) << NatList nList -> {
5        counter++;
6        System.out.println("Positive integer #" + counter + " = " + `pnat);
7      }
8    }
9  }
```

The Java method printPositiveIntegers takes a list nList whose elements are terms of type JNat and print only those built from the operator suc. The alias pnat refers to the subterm suc(y) which is manipulated in the action.

Implicit notation

When defining the algebraic signature of non-variadic operators, each one of their arguments are named. This makes possible to specify a given subterm of a pattern built from a non-variadic operator only by referring to its name. The [ ] construct enables the matching condition to consider specific arguments and ignore the other ones.

Example 2.5. *Consider the handwritten mapping presented in Section 2.2.1 incremented by the following Java class Jplus and its implementation by the algebraic operator plus:*

```
1  static class Jplus extends JNat {
2    public JNat n1;
3    public JNat n2;
4    public Jplus() { }
5    public Jplus(JNat n1, JNat n2) {
6      this.n1 = n1;
7      this.n2 = n2;
```

```
8     }
9   }

11  %op Nat plus(n1:Nat,n2:Nat) {
12    is_fsym(s)     { (s instanceof Jplus) }
13    get_slot(n1,s) { ((Jplus)s).n1 }
14    get_slot(n2,s) { ((Jplus)s).n2 }
15    make(t0,t1)    { new Jplus(t0,t1) }
16  }
```

We can use the Java method printFirstElement to print only the first element of plus:

```
1   public String  printFirstElement (JNat t) {
2     %match {
3       plus[n1=x] << t -> { return printJNat(`x); }
4     }
5     return "";
6   }
```

The Java method printFirstElement takes a term built from plus and considers only its argument named n1 to be printed. The pattern plus[n1=x] is equivalent to plus(x,_) with the advantage of being more concise since only the interesting subterm is (explicitly) pointed. Furthermore, the implicit notation is not impacted by slightly modifications on algebraic signatures such as the order of arguments or the arity of the operators. The auxiliary Java method printJNat converts an algebraic term of type JNat to its string representation.

2.2.4 Strategies

The **%match** construct enables both the deconstruction of algebraic terms and the definition of rewrite rules. In Tom, the application of rewrite rules can be controlled by the use of *strategies* defining which rules must be applied in which positions of terms. The strategy language proposed by Tom is made of a Java library of strategy combinators inspired by the Stratego language [VBT98] which in its turn was inspired by the strategy language of ELAN.

Elementary strategies are created by the **%strategy** construct and correspond to a TRS represented by a Java object of the class Strategy. The rules of the TRS are applied only once in the head position of a given term.

Example 2.6. *Consider the handwritten mapping of Example 2.5. The operator plus can be defined as the addition operator of Peano arithmetic by the following strategy:*

```
1   %strategy addition() extends Identity () {
2     visit  Nat {
3       plus(zero(),  x) -> { return `x; }
4       plus(suc(y),  x) -> { return `suc(plus(y,x)); }
5     }
6   }
```

The strategy addition is compiled as an object of the Java class Strategy whose instance represents the TRS $\mathcal{R}^{pe}$ presented in Example 1.36. The application of addition consists in calling the method <T> T visit(T) of the class Strategy where T is instantiated by Nat. Thus, addition.visit(obj1) corresponds to the application of addition to a given object obj1. Considering obj1 as the representation of the algebraic term plus(suc(zero()),zero()) of type Nat, the second rewrite rule of the strategy is applied and the return value is a new object representing suc(plus(zero(),zero())) also of type Nat. However, the application of addition to another object obj2 representing suc(plus(suc(zero()),zero())) returns the object obj2 unchanged. This happens because the elementary strategy Identity extended by addition is applied since none of the rewrite rules of the TRS apply at head position.

More complex strategies can be defined by the use of two categories of strategy combinators:

- composition combinators that enables the construction of a complex strategy composed of elementary strategies,
- traversal combinators that allows the application of a strategy in any position of a term.

Strategy combinators are Java classes whose objects are strategies and constructors take strategies as arguments. For instance, the composition combinator Sequence is the rewrite relation $\mathcal{S}eq$ formed by the composition of two rewrite relations $\mathcal{S}_1$ and $\mathcal{S}_2$:

$$(\mathcal{S}eq\ \mathcal{S}_1\ \mathcal{S}_2)(t) = (\mathcal{S}_2 \circ \mathcal{S}_1)(t)$$

The relations $\mathcal{S}_1$ and $\mathcal{S}_2$ are defined by the TRS represented respectively by the strategies s1 and s2. Therefore, the application of Sequence(s1,s2) to a term t corresponds to the method call (new Sequence(s1,s2)).visit(t). The return value is the term t'' if there exists some term t' such that s1.visit(t) returns t' and s2.visit(t') returns t''. But otherwise, a VisitFailure exception is thrown.

As an example of traversal combinator we mention All that encodes the rewrite relation $\mathcal{A}ll$ parametrized by another rewrite relation $\mathcal{S}$:

$$(\mathcal{A}ll\ \mathcal{S})(f(t_1,\ldots,t_n)) = f((\mathcal{S})(t_1),\ldots,(\mathcal{S})(t_n))$$

The relation $\mathcal{S}$ is defined by the TRS represented by the strategy s. The application of All(s) to a term t returns a term t' corresponding to the application of s to all arguments of t if all applications succeed. But if at least one application fails, then a VisitFailure exception is thrown.

Strategy combinators can even be combined to each other in order to define other classic complex strategies. For instance, the classic *top-down* strategy is formed by the composition of Sequence and All:

$$(\mathcal{T}opdown\ \mathcal{S})(t) = (\mathcal{S}eq\ \mathcal{S}\ (\mathcal{A}ll\ (\mathcal{T}opdown\ \mathcal{S})))(t)$$

The relation $\mathcal{S}$ is defined by the TRS represented by the strategy s. The application of TopDown(s) to a term t corresponds to the application of s to all subterms of t starting from the head position. Since the strategy All is used, a VisitFailure exception is thrown if at least one application fails. Thus, different from

the single application of addition, (new TopDown(addition)).visit(plus(suc(zero()),zero())) returns an object representing suc(zero()).

Many other complex strategies as the recursive ones are provided by the strategy language of Tom. Emilie Balland has presented this strategy language in details as well as its implementation in her thesis [Bal09].

2.2.5 Compilation in Tom

As explained in Section 2.2.1, mappings between Tom algebraic data types and Java concrete data types can be done manually or automatically. In the first case, handwritten mappings must be provided to Tom as well as all the Java classes implemented by the algebraic data types. In the second case, both mappings and Java class hierarchies can be generated by Gom from a many-sorted signature and then provided to Tom. The compilation process of a Tom program is illustrated in Figure 2.1.

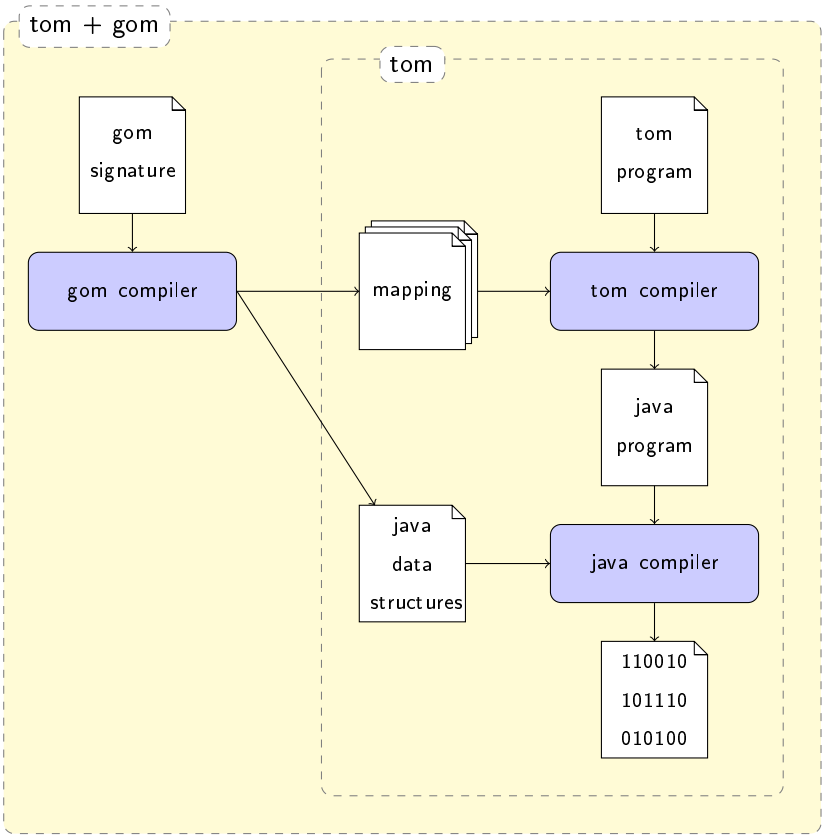


Figure 2.1: The compilation process of a Tom program.

The mapping and the Tom program yielded to the Tom compiler is translated into a Java program which in its turn is translated into Java bytecode by the Java compiler.

2.3 New developments for Tom from this thesis

Tom is a programming language designed to support the analysis and the transformation of various kind of tree-based documents. To help the programmer to specify and implement analysis and transformation tools, Tom provides several high level concepts and constructs such as algebraic signatures, pattern matching, rewrite rules, and strategies to control their application.

The notion of strategy is very important since it allows describing in a simple and efficient way how a set of rules should be applied. Moreover, it describes how a given information could be retrieved or transformed in a tree. To be safe, in a first approximation we consider that a strategy cannot modify the sort of a term, otherwise this could result in an ill-typed term, i.e. a term no longer compatible with the signature on which it is built. Accordingly, an important improvement required by Tom is the adoption of a formal method to ensure that transformations applied by strategies preserve the sort of terms.

This thesis proposes a type system within the scope of the development of Tom as a formal approach for proving the absence of ill-typed terms, while keeping the workload of the programmer as low as possible. Since Tom consists mainly of an extension of Java, the static typing of the Tom constructs — in particular matching constructs — stays compatible with the features of Java. Furthermore, considering a type specification built from algebraic data types, we detect as many type errors in programs and as soon as possible (i.e. still at Tom compilation time) even ignoring concrete data types.

This thesis presents theoretical and practical results about type systems forTom. Its practical Tom impact can be summed up as follows:

Typing The new typer subsumes the previous (relatively weak) typing-related capabilities of Tom. It provides a constraint-based type inference algorithm for the language in order to obtain more safety over types. The types are interpreted as unsorted terms and provide enough information to distinguish between many-sorted terms built from identical variadic operators. Internally, the Gom algebraic signature representing the Tom language was extended to deal with type annotations and with informations about type relation. Also, data constructors for the representation of type constraints were added. The theoretical and practical foundations for those developments are presented in Chapter 3.

Subtyping However, the new typer also handles subtyping while keeping type safety. This principal feature adds for the user the **extends** construct. In combination with the **%typeterm** construct for the definition of type constructors, **extends** enables to specify the inheritance relation between algebraic types when defining a handwritten mapping. The typer was enriched with subtyping as a partial order over types and the constraint language was extended to subtyping constraints in addition to the equality ones. For the resolution of subtyping constraints, we defined rules for a combination of constraint propagation, generation of solution and garbage

collecting of remaining constraints. Specific details concerning this aspect of the work are found in Chapter 4.

Pattern-matching and subtyping The integration of subtyping in Tom also impacted pattern matching and led to the redefinition of the syntactic matching algorithm to handle subtyping constraints. The operational semantics defined in Chapter 3 is extended in Chapter 4 to allow for type inclusion at the pattern level. Type verification for the solution is generated at the Java level, since the fact that Tom is an embedded language prevents it from analyzing its host language. Chapter 5 shows a case study using pattern matching with subtyping in Tom where the language is used for model transformation purposes.

Chapter 3

Typing without subtypes in Tom

As presented in Chapter 2, the development of the Tom language is characterized by the extension of another existing programming language as Java. Naturally, we still expect to perform program analysis on the resulting extended language. For this purpose, in this chapter we introduce a formalization not only of the Tom algebraic terms and types but also of a Tom type system compatible with the Java type system. We consider both type checking and type inference while presenting our static typing approach.

3.1 Motivation

This thesis aims at formalizing and implementing a type system for Tom which improves on the preceding informal one. The foremost achievement of this type system is to ensure the safety of Tom programs: if a Tom program is well-typed, then it must avoid execution errors due to "semantically empty" instructions. In order to understand execution errors, we define an operational semantics for Tom, described as a series of reductions (see Definition 1.20) on the Tom core syntax.

Since Tom is a statically typed language, the type system can perform two kinds of operation:

- type inference, which accepts that some variables and function symbols have no type annotations and looks for a valid type in order to obtain well-typed terms,
- type checking, which requires all variables and function symbols to have type annotations and verifies whether terms are well-typed according to them.

Although dealing with algebraic terms built from typed operators, the Tom core syntax is not completely explicitly typed because some Tom terms such as variables do not have type annotations. For that reason, the Tom type system is described in a "Curry-style". In Curry-style type systems, the information provided by sparse type annotations (in our case the functions are annotated) is often enough for the system to infer the types left unspecified: this is an

added ease of use for the user. We note that type inference as discussed in this chapter can also be referred to as *type reconstruction* since the automated deduction of type is not required for all Tom expressions, but only for a subset of them.

A particularity of the type inference system we are to introduce is to be based on type constraints which brings two main advantages. The first one is that type annotations are type-checked since they are used in the generation and subsequent resolution of constraints. This discards the necessity of additional type checking. The second advantage of the constraint-based approach is that the addition of new kind of constraints (e.g. subtyping constraints) would only require a modification of the type inference rules and the rewrite rules for respectively generation and resolution of constraints.

3.2 Related Works

Embedded languages allow a host language to be extended by adding elements of a given DSL. For example, Tom provides pattern matching and rule based constructs for Java. There are also many embedded languages that propose the integration of features of others programming paradigms or even elements for scripting, i.e. for configuring and controlling application programs in the host language. While some general purpose embedded languages provide static typing, most of the scripting languages are dynamically type checked or do not address type safety issues. For instance, the following embedded languages have type systems with different degrees of type safety:

Lua The Lua language [IdFF96] is mainly used as a scripting language or an extension to C (or any other language with interface to C). Lua provides an easy-to-use and tiny C API used by its standalone Lua interpreter written in C. Because Lua offers a small set of general features, it strives to be extended to fit different kind of applications. Lua is a dynamically typed language designed expressly for embedding. It provides some built-in types and facilities to define algebraic data types. In Lua, values have types, variables are typeless placeholders for values of any type and type checking is performed at runtime.

BeanShell BeanShell [Nie02] is a scripting language for Java and can be interactively used for Java experimentation and debugging. The BeanShell interpreter is loosely typed by default, performing type checking at runtime when appropriate. However, it can be set to support only strongly typed macros where typed variables are declared. In BeanShell, variables can be typed or untyped. Typed variables are considered local to the scope in where they are declared. Untyped variables in their turn can hold Java primitive values and act as if they were declared outside of their enclosing scope. Similarly, methods with dynamic arguments and return types have their types dynamically determined when these methods are executed.

Jess Jess [Hil01, Hil03] is both an expert system shell running as a standalone program and a scripting language for Java working embedded in Java code. Indeed, Jess is a Java API acting as a Jess interpreter that can also be used as a rule engine, i.e. a program that applies rules to data of a knowledge base. The Jess rules are activated whenever their left-hand sides are satisfied. Keeping a list of rules, Jess performs pattern matching after each insertion of a data in the knowledge base, but activated rules are fired only when the rule engine is running. There are three kinds of data in Jess where one of them, called *unordered facts*, allows data to have their types specified. However, these types accept only a limited number of primitive predefined values and no type checking is provided by Jess.

AspectJ AspectJ [KHH+01] is an extension of Java which adds aspect-oriented features allowing the separation of concerns of Java programs. Concerns are encapsulated by *aspects* that are special constructs provided by AspectJ. Aspects can be added to Java code without modifying the original code, and then making all valid Java code also valid AspectJ code. The static structure of a Java program can be modified through the use of *inter-type declarations* which are entities of an aspect. This enables the declaration of members and supertypes of Java classes in an aspect. AspectJ code is compiled into standard Java bytecode which is subject to the static type checking performed by the Java type system.

As in dynamically typed languages, statically typed languages may not require type declarations and instead perform type inference for terms. The idea of finding types for untyped terms through resolution of systems of type constraints was first considered by Roger Hindley [Hin69] who represented terms in combinatory logic [CF58], that is, terms consist only of functions with no free variables, called *combinators*, combined by simple juxtaposition. This work was later independently developed by Robin Milner [Mil78] who extended it to support *parametric polymorphism* in λ-calculus, that is, the abstraction of base types through the use of type variables. He also proved the soundness and correctness of the type system. Type inference *à la* Hindley-Milner is often used for the functional paradigm. It also constitutes the basis of rewriting languages as those presented in this section. They are statically typed and support programming with algebraic data type and rewriting systems.

3.2.1 Maude

Maude [CDE+07] is a language based on the membership equational logic introduced by Adel Bouhoula, Jean-Pierre Jouannaud and José Meseguer [BJM00, Mes97]. Membership equational logic is a sublogic of the rewriting logic defined by José Meseguer [Mes92] and extends equational logic composed of equalities t = t' by membership axioms t : *s*, stating that a term t has a certain sort s. Maude incorporates most features of seminal works done on OBJ [FGJM85, GWM+93]. This conveys the definition of algebraic data types and operators in functional

modules corresponding to equational theories where operator overloading, dependent types, parameterized modules and multiple inheritance are allowed. In Maude, all variables and operators are type annotated and types are divided into two categories: *sorts*, which are names for the data types, and *kinds*, which are equivalence classes of the sorts. Moreover, Maude supports equational theories $\mathcal{A}$, $\mathcal{AU}$, $\mathcal{AC}$ and $\mathcal{ACU}$ and assume that they are confluent and terminating. These theories are declared as equational attributes and only apply to binary operators following specific requirements about sorts of their ranks. For instance, for the $\mathcal{AU}$ theory, both domain and codomain sorts must belong to the same kind. Accordingly, we can define the handwritten mapping presented in Section 2.2.1 in a ExamplePeano module which does not consider subsort relation:

```
1 fmod EXAMPLEPEANO is
2   sort Nat .
3   op zero : -> Nat .
4   op suc : Nat -> Nat .
5   op empty : -> Nat .
6   op concNat : Nat Nat -> Nat [assoc id: empty] .
7 endfm
```

Here, Maude differs from Tom: the associativity property and the neutral element for the binary operator concNat are declared explicitly in order to enable pattern matching modulo $\mathcal{AU}$ over terms built from this operator. Then, terms built from variadic operators in Tom correspond to terms composed of nested associative binary operators and subsequently flattened in Maude. However, the Tom-specific difficulty in discerning an insertion of a term whose codomain sort belongs to a (possible) different kind from a concatenation of terms (modulo flattening) does not exist in Maude. Consequently, the Maude type system type-checks terms built from either associative or non-associative function operators in the same way.

3.2.2 Scala

The Scala programming language [Ode11, Cre06] is a blend of two consolidated programming paradigms: functional and object-oriented programming, working seamlessly with Java and C#. It is a strongly statically typed language whose type inference is done locally based on type annotations and supporting structural and existential types. It also supports features of functional programming such as anonymous functions, high-order functions, currying and pattern matching. In connection with the object-oriented paradigm, functions are also objects being manipulated as values and algebraic data types are described by classes and *traits*. Classes are extended by subclassing and their methods can be overloaded and overridden by subclasses. Classes can import code from one or more *traits* which are interfaces that may contain the same code as classes. Besides multiple inheritance, Scala also supports parameterized classes and parameterized methods. Type constructors correspond to abstract classes with

case classes as data constructors. Case classes can be ordered in class hierarchies but can not inherit from other case classes. To illustrate the definition of algebraic data types through abstract and case classes, we present a Scala code corresponding to the handwritten mapping of Section 2.2.1:

```
1  abstract class Nat
2  case class zero() extends Nat
3  case class suc(n: Nat) extends Nat

5  abstract class NatList
6  case class concNat(m: List[Nat]) extends NatList
```

In line 6, we use a first-order type constructor List representing an ordered collection of elements (of data type Nat in our case). Another possibility to define a variadic data constructor is the use of the suffix * representing a sequence argument:

```
case class concNat(m: Nat*) extends NatList
```

The type annotation _* can be used to mark the last argument of terms built from variadic constructors. Terms built from constructors defined by case classes can be deconstructed by pattern matching whose operational semantics is defined in Burak Emir's thesis [Emi07]. In Scala, patterns are linear, i.e. a same variable can not occur more than once in a pattern, and anonymous variables _ and alias introduced by @ are allowed. The pattern matching provided by Scala is syntactic even for patterns built from variadic constructors, since, in contrast to Tom, matching modulo equational theories is not supported. Accordingly, the Example 2.4 can be written in Scala as in the following:

```
7  def printPositiveIntegers (nList: NatList, countSuc: Int) {
8    var counter = countSuc
9    nList match {
10     case concNat(Nil) => {}
11     case concNat(x::z) => {
12       x match {
13         case pnat@suc(y) => {
14           counter = counter + 1
15           println ("Positive integer #" + counter + " = " + pnat)
16         }
17         case _ => {}
18       }
19       printPositiveIntegers (concNat(z),counter)
20     }
21   }
22 }
```

Another difference between pattern matching in Tom and Scala is with respect to their semantics. In Scala, the patterns are tried in sequence until one is found which matches the subject. The associated action is executed and the **case**

is complete. This leads to the use of recursion in order to perform matching over each element of a collection, as at line 19. Still concerning pattern matching, Scala performs exhaustive check for it which explains the use of an anonymous variable as pattern at line 17. A local type inference for patterns constructs a set of type constraints over type variables as in Tom (see Section 3.3.4) where variables of patterns are bound in the right-hand side, i.e. in the block code. Moreover, for patterns built from constructors that are of a parametrized type, constraints possibly limit the type variables occurring in the type parameters.

3.2.3 ELAN

ELAN [BKK+98, BCD+06] is a rule based programming language created and implemented by the Protheo group, to which our research team is a follow-up. As Maude, ELAN also has its foundations in the rewriting logic defined by José Meseguer [Mes92] and incorporates features of OBJ [GWM+93]. However, the notion of strategy jointly with the one of rewrite rule constitutes the first class concept of the language allowing to express non-deterministic computations. Sorts and operators are introduced by signatures in an algebraic-like specification formalism where the equational theory $\mathcal{AC}$ is supported for binary operators with the same sort for domain and codomain. In addition to equational attributes, information about precedence and syntactic (left or right) associativity of operators can be declared for parsing purposes. The algebraic types and operators of the handwritten mapping of Section 2.2.1 can be defined in ELAN as follows:

```
1  module examplepeano
2    sort
3      Nat NatList;
4    end

6    operators  global
7      zero              :                      Nat;
8      suc(@)            : (Nat)                Nat;
9      @                 : (Nat)                NatList;
10     nil               :                      NatList;
11     @.@               : (NatList NatList)    NatList assocRight;
12     concNat(@)        : (NatList)            NatList;
13   end
```

Although ELAN does not support subsort declarations, it allows the use of anonymous operators `@` to define injective functions from a sort to another one, as at line 9. The anonymous operator is internally represented by a constructor with a fresh name and whose domain and codomain are respectively Nat and NatList. In association with it, the definition of a concatenation operator (at line 11) allows the operator concNat to be used as a variadic operator. Since ELAN also supports parameterized types, another possibility to simulate the indefinite arity of concNat is to use the type constructor List provided by the ELAN standard

library:

```
concNat(@)      : (List[Nat])        List[Nat];
```

The semantics of the operators is described by a set of unlabelled rewrite rules which are systematically applied on terms. Accordingly, we use unlabelled rules to declare the flattening property (see Definition 1.29) of the operator concNat:

```
14  rules for NatList
15    x   : Nat;
16    y,z : NatList;
17  global
18    [ ] concNat(z.nil)          => concNat(z)    end
19    [ ] concNat(nil.z)          => concNat(z)    end
20    [ ] concNat(y.concNat(z))   => concNat(y.z)  end
21    [ ] concNat(concNat(y).z)   => concNat(y.z)  end
22    [ ] concNat(concNat(z))     => concNat(z)    end
23  end
```

Another kind of rewrite rules are the labeled ones considered as elementary strategies whose application is controlled by strategy combinators and returns all results found. Strategy combinators are built-in strategy operators provided by the ELAN strategy language initially proposed by Marian Vittek et al. [Vit94, BKK+97] and lately extended in Peter Borovanský's thesis [Bor98]. In this way, we also define a labeled rule tail to iterate over all elements of a term built from concNat and two labeled rules with same name head to select those elements which occur in the first position of the term and are built from suc:

```
23  rules for NatList
24    x   : Nat;
25    z   : NatList;
26  local
27    [head] concNat(suc(x).z)  => suc(x)       end
28    [head] concNat(suc(x))    => suc(x)       end
29    [tail] concNat(x.z)       => concNat(z)   end
30  end
```

Then we introduce a constant strategy operator to encode the method printPositiveIntegers of Example 2.4:

```
31  strategies for NatList
32  implicit
33    [.]  printPositiveIntegers  => iterate*(tail);head  end
34  end
35 end
```

The strategy rule printPositiveIntegers controls the application of the labeled rules by combining the sequence operator ; with the strategy operator iterate*

which repeats the application of its argument strategies until it fails and returns all intermediate results. The application of the strategy rule on a term is done by syntactic matching except for those terms built from $\mathcal{AC}$ operators. For pattern matching modulo $\mathcal{AC}$, all solutions are computed in a multiset from where one of the solutions is chosen and the remaining ones are stored for (possible) future backtracking.

Regardless of not being an embedded language, ELAN strongly inspired the Tom language especially with respect to its rewrite strategies. However, it incorporates a relatively simple type system that concerns terms built from first-order operators but does not perform type inference. As in Maude, this simplicity comes from the restrictions on arity and domain and codomain sorts of signatures of associative-commutative operators.

3.3 The type system

To talk rigorously about type systems and their properties, we need to introduce the notion of *type* used to classify terms of a language. We also need to present formally the Tom core syntax in which we are interested. The evaluation of expressions built from this syntax is afterward described by an operational semantics. Then, we describe our type checking system as a proof calculus for typing judgments about type annotated expressions composing a Tom program. For expressions built from variables having no type information, we define a type inference system as an algorithm to calculate their unspecified type annotations. The resulting formalization allows us to reason about the two aspects of the logic of programming languages: syntax and semantics.

3.3.1 Tom terms and types

As seen in Section 2.2, the Tom algebraic terms are built from variables, syntactic operators and variadic operators. To represent them we introduce respectively the sets $\mathcal{V}$, $\mathcal{F}$ and $\mathcal{F}^\star$. We recall that terms built from variadic operators are often called *lists*. We also recall that *star variables* can be used to represent sublists occurring in patterns of a list-matching construct.

Because Tom terms are built from many-sorted variadic signatures, we introduce a set $\mathcal{S}$ of base sorts represented by Tom algebraic data types. At this stage, we can still build terms from syntactic and variadic operators being of the exact same sort. However, we need to distinguish those two kinds of terms during type checking and type inference. For instance, let us consider a variadic operator $\mathsf{v} \in \mathcal{F}^\star$, two variables $\mathsf{x}, \mathsf{y} \in \mathcal{V}$ and an operator $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^\star$. The insertion of $\mathsf{g}(\mathsf{x})$ into a list $\mathsf{v}(\mathsf{y})$ can lead to two different results depending on the status of g:

1. suppose $\mathsf{g} \in \mathcal{F}^\star$ and $\mathsf{g} = \mathsf{v}$, then, by flattening (see Definition 1.29), the result is a concatenation of both lists, i.e. $\mathsf{v}(\mathsf{x},\mathsf{y})$;

2. suppose $\mathsf{g} \in \mathcal{F}^\star$ and $\mathsf{g} \neq \mathsf{v}$ or if $\mathsf{g} \in \mathcal{F}$, then the result is an insertion of an element into a list, i.e. v(g(x),y).

Hence, we are interested in distinguishing the lists with the same head symbol from the other terms. We introduce the notion of sorts decorated with function symbols to classify terms, for instance s^g with $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^\star$. We also define a special symbol ? for the cases in which the decoration is unimportant, for example, when specifying the domain of a function symbol. Decorated sorts are also called *ground types* since they are the simplest types with whom a term can be classified. We introduce *type variables* as placeholders for ground types. We also introduce a special ground type wt to classify *well-typed* expressions which are not terms, for instance matching conditions, backquote constructs among others. On the whole, the set of types $\mathcal{T}y$ is composed of unsorted terms built from a set of type variables, from $\{wt\}$ and from the set of decorated sorts, i.e. the combination of $\mathcal{S}$ and $\mathcal{F} \cup \mathcal{F}^\star \cup \{?\}$.

Definition 3.1 (Types). *Let wt be a special constant and $\mathcal{X}$ be a countably infinite set of* type variables *denoted by α, β, etc. The set $\mathcal{T}y$ of types is made up of the unsorted terms defined by the following algebraic grammar:*

$$\tau ::= \alpha \mid s^h \mid wt$$

where $\tau \in \mathcal{T}y$, $\alpha \in \mathcal{X}$, $s \in \mathcal{S}$, $h \in \mathcal{F} \cup \mathcal{F}^\star \cup \{?\}$.

A ground type *is a type τ with no variables, i.e. $\mathcal{V}ar(\tau) = \emptyset$.*

The equality of decorated sorts consists in the syntactic equality of sorts enriched by a special comparison of their decorations.

Definition 3.2 (Equality of decorated sorts). *Let $s_1, s_2 \in \mathcal{S}$ be two sorts and $h_1, h_2 \in \mathcal{F} \cup \mathcal{F}^\star$. The* equality of decorated sorts*, denoted by $=_t$, is defined as follows:*

1. $s_1^? =_t s_2^? \Leftrightarrow s_1 = s_2$
2. $s_1^{h_1} =_t s_2^? \Leftrightarrow s_1 = s_2$
3. $s_1^? =_t s_2^{h_2} \Leftrightarrow s_1 = s_2$
4. $s_1^{h_1} =_t s_2^{h_2} \Leftrightarrow s_1 = s_2 \wedge h_1 = h_2$

where $=$ stands for the syntactic equality.

We write $s_1^{h_1} \neq_t s_2^{h_2}$ to mean that $\neg(s_1^{h_1} =_t s_2^{h_2})$.

The notion of types is really useful for the typing of Tom expressions. For this reason, we define a correspondence called *decoration cleaning* between types and base sorts which allows types to be converted into non-decorated sorts. We mainly use this when representing the explicit type annotation of the **%match** construct through a type.

Definition 3.3 (Decoration cleaning). *The function $|\cdot|$ is a partial function from $\mathcal{T}y$ to $\mathcal{S}$ defined by:*

$$|s^h| = s$$

where $s \in \mathcal{S}$ *and* $h \in \mathcal{F} \cup \mathcal{F}^\star \cup \{?\}$.

Considering the set $\mathcal{T}y$ of types, the sets $\mathcal{F}$ and $\mathcal{F}^\star$ of function symbols and a countably infinite set $\mathcal{V}$ of variables, Tom terms are built from the term algebra $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ (see Definition 1.26). The set of Tom ground terms is denoted by $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$.

Definition 3.4 (Tom terms)**.** *The language of Tom terms is specified by the following algebraic grammar:*

$$term ::= \mathsf{x} \mid \mathsf{x*} \mid \mathsf{f}(\underbrace{term, \ldots, term}_{n}) \mid \mathsf{g}(term,\ldots,term)$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}_n$ *and* $\mathsf{g} \in \mathcal{F}^\star$.

A Tom code can contain Java code which in its turn can contain Tom code and so on. Since we intend to check and infer types of Tom terms, we consider an abstraction consisting only of expressions involving Tom terms. Thus, Java variables bound to Tom terms are represented by variables of $\mathcal{V}$. In addition, Java functions manipulating Tom terms or being manipulated by Tom terms are entirely represented by Tom terms. The remaining Java objects are dismissed.

Definition 3.5 (Tom core abstract syntax)**.** *The* Tom core abstract syntax *is specified by the following algebraic grammar:*

$$
\begin{array}{lcl}
\mathit{code} & ::= & \{\mathit{rule};\ldots;\mathit{rule}\} \mid \text{`}\mathit{term} \mid \mathsf{x} := \text{`}\mathit{term} \\
\mathit{rule} & ::= & \mathit{cond} \longrightarrow \mathit{action} \\
\mathit{cond} & ::= & \mathit{pattern} \ll |\tau|\ \mathit{term} \mid \mathit{term} \diamond \mathit{term} \mid \mathit{cond} \wedge \mathit{cond} \mid \mathit{cond} \vee \mathit{cond} \\
\mathit{pattern} & ::= & \mathsf{x} \mid \mathsf{x*} \mid \mathsf{g}(\mathit{pattern},\ldots,\mathit{pattern}) \mid \mathsf{x}@\mathit{pattern} \mid !\mathit{pattern} \\
\mathit{term} & ::= & \mathsf{x} \mid \mathsf{x*} \mid \mathsf{g}(\mathit{term},\ldots,\mathit{term}) \\
\mathit{action} & ::= & \mathit{code} \mid (\mathit{action};\ldots;\mathit{action})
\end{array}
$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^{\star}$ *and* $\tau \in \mathcal{T}y \backslash \{wt\}$.

Terms built from this grammar are called Tom expressions.

A Tom *code is a set of code.*

A *code* can be an assignment instruction, a backquoted term or a list of rules *cond* $\longrightarrow$ *action*. The assignment of a term t to a variable x is represented by x := `t. The left-hand side of a rule is either a single condition or a conjunction/disjunction of single conditions. A single condition is either a *matching* or a *numeric condition*. *Matching conditions* are match-equations $\mathsf{t}_1 \ll^{?} |\tau|\ \mathsf{t}_2$ (see Definition 1.37 in where we omit the symbol "?" for the sake of readability. τ denotes a type and $|\tau|$ is its undecorated form (see Definition 3.3). We call *anti-pattern* the patterns composed of one or more !*pattern*. *Numeric conditions* are conditions of the form $\mathsf{t}_1 \diamond \mathsf{t}_2$ where $\diamond$ stands for the representation of the logical operators $=$, $\neq$, $<$, $\leq$, $>$ and $\geq$. The right-hand side of a rule is possibly composed of one or more *code*. Thus, an action can be a sequence of actions, i.e. a sequence of list of rules and/or backquoted terms. A `*term* corresponds to the backquote construct allowing a Tom term to be built into a block of Java code. Moreover, anonymous variables _ introduced in Section 2.2.2 do not appear in the grammar since they are represented by *fresh* variables whose name is not used elsewhere.

Example 3.6. *Consider the* Java *method* printPositiveIntegers *of Example 2.4. The* **%match** *construct is represented by the following* Tom *code:*

$$\{\mathsf{concNat(x*,pnat@suc(y),z*)} \ll |\mathit{NatList}^{?}|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}$$

We illustrate the abstract representation of a Tom program composed of Tom code and Java code by means of an example with nested *block* and Tom terms occurring in Java statements.

Example 3.7. *Consider the handwritten mapping presented in Section 2.2.1 and the* Java *method* leq *introduced in Example 2.3. The following* Java *method describes a sorting algorithm for the insertion of an element into a list:*

```
public JNatList insertionSort(JNat t, JNatList nList) {
  %match {
    concNat()           << NatList nList -> { return `concNat(t); }
    concNat(head,tail*) << NatList nList -> {
      %match {
```

```
          true  << boolean leq(t,head) -> { return `concNat(t,nList*); }
          false << boolean leq(t,head) -> {
            JNatList nSubList = `insertionSort(t, tail*);
            return `concNat(head,nSubList*);
          }
        }
      }
    }
    throw new RuntimeException("Error found in insertionSort.");
  }
```

The method insertionSort *is represented in* Tom *core abstract syntax as follows:*

```
{
  concNat() ≪ |NatList?| nList ⟶ `concNat(t);
  concNat(head,tail*) ≪ |NatList?| nList ⟶
  ({
    true ≪ |boolean?| leq(t,head) ⟶ `concNat(t,nList*);
    false ≪ |boolean?| leq(t,head) ⟶ (nSubList :=`insertionSort(t,tail*);
`concNat(head,nSubList*))
  })
}
```

Throughout this chapter, we use the metavariable t and variants such as t_1, t_2 and t_n to stand for Tom terms. The metavariables are not variables of the Tom language, but variables that can be instantiated by a particular Tom expression. Moreover, metavariables f and v denote respectively syntactic operators and variadic operators while g stands for operators which can be syntactic or variadic. The metavariable h is used as sort decoration and stands for f, v, g or ?. An entire summary of metavariable conventions can be found in Appendix B.

3.3.2 Operational semantics

The operational semantics of a rewriting language like Tom can be described by using formalisms with rewriting capabilities such as the *ρ-calculus* [CK01a, CK01b]. The ρ-calculus also called *rewriting calculus* is a framework featuring λ-calculus and pattern matching on non-normalizing terms. Its syntax and semantics is described by Horatiu Cirstea, Luigi Liquori and Benjamin Wack [CLW03] among others. It has been used as an operational semantics for ELAN where the ρ-encoding of conditional rewriting was extended to support conditional rewrite rules with and without local assignments as explained in the work of Horatiu Cirstea and Claude Kirchner [CK01b]. The ρ-calculus representation of ELAN rules also handles non-determinism and could be well suitable to Tom rules with left-hand sides composed of conjunctions/disjunctions of matching and numeric conditions. However, Tom rules are not purely rewriting rules in the sense that Java statements can occurs at their right-hand sides compromising their representation.

In order to enjoy a higher level of abstraction than that offered by the ρ-calculus, we focus on another well-known approach to semantics where some

operational aspects can be partially ignored. We briefly describe the big-step operational semantics of the Tom language according to the formalism defined by Gilles Kahn [Kah87]. The work of Didier Rémy [RÓ2] also inspired the syntactic description of Tom code evaluation which is defined by a reduction relation on expressions, written $e \Downarrow e'$. We introduce the notion of *environment* to register the evaluation state of a code, i.e. the *values* of all its current variables.

Definition 3.8 (Values). *The set $\mathcal{V}al$ of* values *is composed of the set of ground terms $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ with the addition of two distinguished constants* accept *and* reject*:*

$$\dot{v} ::= \dot{u} \mid \mathsf{accept} \mid \mathsf{reject}$$
$$\dot{u} ::= \mathsf{g}(\dot{u},\ldots,\dot{u})$$

where $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^\star$, $\dot{u} \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ and $\dot{v} \in \mathcal{V}al$.

We denote by $\flat(\dot{u})$ the flattening operation applied to the value $\dot{u}$ (see Definition 1.29).

Definition 3.9 (Environment). *An (evaluation)* environment *ρ is a partial function from $\mathcal{V}$ to $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$. Let $\mathsf{x} \in \mathcal{V}$ and $\mathsf{t} \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$. We denote by $\rho[\mathsf{x} \mapsto \mathsf{t}]$ the environment ρ' such that, for all $\mathsf{y} \in \mathcal{V}$ such that $\mathsf{y} \in \mathcal{D}om(\rho)$:*

$$\rho'(\mathsf{y}) = \begin{cases} \mathsf{t} & \text{if } \mathsf{x} = \mathsf{y} \\ \rho(\mathsf{y}) & \text{otherwise} \end{cases}$$

We denote by $\mathcal{E}nv$ the set of all environments.

We define an overloading operator in order to avoid variable clash while applying union to two environments.

Definition 3.10 (Overloading). *An* overloading *operation $\rho_1 \lhd \rho_2$ of two environments ρ_1 and ρ_2 is defined by:*

$$\rho_1 \lhd \rho_2(\mathsf{x}) = \begin{cases} \rho_1(\mathsf{x}) & \text{if } \mathsf{x} \notin \mathcal{D}om(\rho_2) \\ \rho_2(\mathsf{x}) & \text{otherwise} \end{cases}$$

When dealing with more than one environment, the notion of semantic compatibility is useful to ensure that a same variable occurring in these environments is mapped to a same value.

Definition 3.11 (Semantic compatibility). *The* semantic compatibility *of two environments ρ_1 and ρ_2, written $\rho_1 \bowtie \rho_2$ is defined by:*

$$\forall \mathsf{x}\ (\mathsf{x} \in \mathcal{D}om(\rho_1) \wedge \mathsf{x} \in \mathcal{D}om(\rho_2) \ \Leftrightarrow \rho_1(\mathsf{x}) = \rho_2(\mathsf{x}))$$

We write $\rho_1 \not\bowtie \rho_2$ to mean that ρ_1 and ρ_2 are not semantically compatible.

An evaluation judgment for Tom expressions has the form $\rho \Vdash \mathsf{t} \Downarrow (\rho', \dot{v})$ or $\rho \Vdash \mathsf{t} \Downarrow (\{\rho'_1, \ldots, \rho'_n\}, \dot{v}')$ where $\{\rho'_1, \ldots, \rho'_n\}$ represents a set of environments and $\dot{v}' \in \mathcal{V}al \backslash \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$. The inference rules for natural semantics of Tom are described in Figures 3.1, 3.2, 3.3 and 3.4. They are considered modulo renaming of variables. We assume that matching and numeric conditions over terms built from variadic operators are considered modulo flattening (see Definition 1.29). Matching conditions depend on judgments describing acceptance or rejection of patterns. List-matching is a special form of matching modulo $\mathcal{AU}$ where pattern and subject have the same head symbol. The evaluation of list-matching is described here in an "abstract way" while its detailed description consists in the small-step semantics formalism defined by Gordon Plotkin [Plo81, Plo04]. We adopt the small-step semantics of matching modulo $\mathcal{AU}$ as well as the one of anti-matching for the Tom language described by Radu Kopetz in chapter 3 of his thesis [Kop08]. Moreover, a discussion about unification and matching modulo $\mathcal{AU}$ among other equational theories can be found in the work of Jean-Pierre Jouannaud and Claude Kirchner [JK91]. Numeric conditions depend on judgments describing acceptance or rejection of comparisons between terms.

$$\frac{\mathsf{x} \in \mathcal{D}om(\rho) \quad \rho(\mathsf{x}) = \dot{u}}{\rho \Vdash \mathsf{x} \Downarrow (\rho, \dot{u})}\ \text{E-Var} \qquad \frac{}{\rho \Vdash \dot{v} \Downarrow (\rho, \dot{v})}\ \text{E-Val}$$

$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \ldots \quad \rho \Vdash \mathsf{t}_n \Downarrow \dot{u}_n \quad \mathsf{f} \in \mathcal{F}_n}{\rho \Vdash \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \Downarrow (\rho, \mathsf{f}(\dot{u}_1, \ldots, \dot{u}_n))}\ \text{E-Fun}$$

$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \ldots \quad \rho \Vdash \mathsf{t}_n \Downarrow \dot{u}_n \quad \mathsf{v} \in \mathcal{F}^\star}{\rho \Vdash \mathsf{v}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \Downarrow (\rho, \flat(\, \mathsf{v}(\dot{u}_1, \ldots, \dot{u}_n)))}\ \text{E-List}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad \rho \Vdash \mathsf{t} \Downarrow (\rho, \dot{u})}{\rho \Vdash (\mathsf{x} := \text{`}\mathsf{t}) \Downarrow (\rho \lessdot \{\mathsf{x} \mapsto \dot{u}\}, \mathsf{accept})}\ \text{E-Ass} \qquad \frac{\rho \Vdash \mathsf{t} \Downarrow (\rho, \dot{u})}{\rho \Vdash \text{`}\mathsf{t} \Downarrow (\rho, \mathsf{accept})}\ \text{E-BqTerm}$$

where $\mathsf{t}, \mathsf{t}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ and $\dot{u}, \dot{u}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ for $i \in [1, n]$.

Figure 3.1: Evaluation rules for (backquoted) terms and assignments.

In Figure 3.1, the rule [E-Var] describes the evaluation of a variable bound to a value in the environment: the variable reduces to this value without modifying the environment. For the evaluation of a value, the rule [E-Val] returns the original environment and value. Rules [E-Fun] and [E-List] correspond to the unfolding of syntactic and variadic operators, respectively. Moreover, the rule [E-List] considers flat lists, i.e. lists with no nested sublists of a same head operator. The result is obtained by evaluating each argument of the function. For the evaluation of an assignment instruction, the rule [E-Ass] increments

the environment by setting the value of the variable to the result of evaluating the backquoted term on the right side of the operator :=. Rule [E-BqTerm] describes evaluation of Tom terms occurring in Java code. As an approximative semantics of Java terms combined with Tom terms, we consider only the cases where that Java variables or Java functions represented originally by Tom terms only evaluates to Tom ground terms. If this property does not hold, the Java compiler is expected to raise a compilation error.

Rules whose names are prefixed by "E-M" stand for the semantics of pattern matching and produce a non-empty set of environments. Types τ occurring in matching conditions correspond to the explicit type annotation of which the pattern is expected to be. The type of a term t is obtained by an auxiliary judgment typeof([ ],t) which will be discussed in details in Section 3.3.3 and where [ ] stands for an empty sequence. Rules [E-MVarAcc] and [E-MVarRej] describe respectively acceptance and rejection for patterns composed of variables whose types must be equal to those of subjects:

$$\frac{\mathsf{x} \in \mathcal{V} \quad \tau =_t \mathsf{typeof}([\,],\mathsf{x}) \quad \tau =_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x} \ll |\tau|\ \dot{u} \Downarrow (\{\rho \lhd \{\mathsf{x} \mapsto \dot{u}\}\}, \mathsf{accept})}\ \text{E-MVarAcc}$$

and

$$\frac{\mathsf{x} \in \mathcal{V} \quad \tau \neq_t \mathsf{typeof}([\,],\mathsf{x}) \vee \tau \neq_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x} \ll |\tau|\ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MVarRej}$$

Similarly, rules [E-MSVarAcc] and [E-MSVarRej] work for patterns composed of star variables and matching lists:

$$\frac{\mathsf{x} \in \mathcal{V} \quad \mathit{symb}(\dot{u}) \in \mathcal{F}^\star \quad \tau =_t \mathsf{typeof}([\,],\mathsf{x}) \quad \tau =_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x}* \ll |\tau|\ \dot{u} \Downarrow (\{\rho \lhd \{\mathsf{x} \mapsto \dot{u}\}\}, \mathsf{accept})}\ \text{E-MSVarAcc}$$

and

$$\frac{\mathsf{x} \in \mathcal{V} \quad \mathit{symb}(\dot{u}) \in \mathcal{F}^\star \quad \tau \neq_t \mathsf{typeof}([\,],\mathsf{x}) \vee \tau \neq_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x}* \ll |\tau|\ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MSVarRej}$$

For patterns composed of variables as alias, the following rule applies propagating the evaluation of the matching problem:

$$\frac{\mathsf{x} \in \mathcal{V} \quad \rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\rho', \dot{v})}{\rho \Vdash \mathsf{x@t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho' \lhd \{\mathsf{x} \mapsto \dot{u}\}\}, \dot{v})}\ \text{E-MAlias}$$

The rules [E-MFunAcc] and [E-MFunRej] apply to patterns built from a same syntactic operator and decomposes their evaluation into the evaluation of pattern matching on their arguments:

$$\frac{\rho_0 \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \dot{u}_1) \Downarrow (\rho_1,\mathsf{accept}) \quad \ldots \quad \rho_0 \Vdash (\mathsf{t}_n \ll |\tau_n|\ \dot{u}_n) \Downarrow (\rho_n,\mathsf{accept}) \quad \mathsf{f} \in \mathcal{F} \quad \tau =_t \mathsf{typeof}([\,],\mathsf{f}(\dot{u}_1,\ldots,\dot{u}_n)) \quad \forall i \in [1,n], \rho_{i-1} \bowtie \rho_i}{\rho_0 \Vdash \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n) \ll |\tau|\ \mathsf{f}(\dot{u}_1,\ldots,\dot{u}_n) \Downarrow (\{\bigcup_{j=0}^{n} \rho_j\}, \mathsf{accept})}\ \text{E-MFunAcc}$$

and

$$\frac{\rho_0 \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \dot{u}_1) \Downarrow (\rho_1,\dot{v}_1) \quad \ldots \quad \rho_0 \Vdash (\mathsf{t}_n \ll |\tau_n|\ \dot{u}_n) \Downarrow (\rho_n,\dot{v}_n) \quad \mathsf{f} \in \mathcal{F} \quad \exists i \in [1,n], (\dot{v}_i = \mathsf{reject} \vee \rho_{i-1} \not\bowtie \rho_i)}{\rho_0 \Vdash \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n) \ll |\tau|\ \mathsf{f}(\dot{u}_1,\ldots,\dot{u}_n) \Downarrow (\{\rho_0\}, \mathsf{reject})}\ \text{E-MFunRej}$$

The rule [E-MFunListRej] applies to both syntactic and variadic operators:

$$\frac{\mathsf{g},\mathsf{g'} \in \mathcal{F} \cup \mathcal{F}^\star \quad \mathsf{g} \neq \mathsf{g'} \vee \tau \neq_t \mathsf{typeof}([\,],\mathsf{g'}(\dot{u}_1,\ldots,\dot{u}_n))}{\rho \Vdash \mathsf{g}(\mathsf{t}_1,\ldots,\mathsf{t}_n) \ll |\tau|\ \mathsf{g'}(\dot{u}_1,\ldots,\dot{u}_n) \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MFunListRej}$$

Its result depends on the equality of both function symbols and types of pattern and subject. All the rules describing acceptance and rejection of syntactic matching are summarized in Figure 3.2 which includes a rule [E-MSubj] applicable to subjects containing variables.

Some operational semantics rules are not syntax directed since the terms involved in their premises are not explicitly linked to those of conclusion. For instance, the rules [E-MListAcc] and [E-MListRej] describe the evaluation of a list-matching:

$$\frac{\rho \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \mathsf{t'}_1) \Downarrow (\rho_1, \mathsf{accept}) \quad \rho_1 \Vdash (\mathsf{t}_2 \ll |\tau_2|\ \mathsf{t'}_2) \Downarrow (\rho_2, \mathsf{accept}) \quad \rho \bowtie \rho_1 \quad \rho_1 \bowtie \rho_2 \quad \mathit{symb}(\mathsf{t}) \in \mathcal{F}^\star \vee \mathit{symb}(\mathsf{t'}) \in \mathcal{F}^\star}{\rho \Vdash \mathsf{t} \ll |\tau|\ \mathsf{t'} \Downarrow (\{\rho_2\}, \mathsf{accept})}\ \text{E-MListAcc}$$

and

$$\frac{\rho \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \mathsf{t'}_1) \Downarrow (\rho_1,\dot{v}_1) \quad \rho_1 \Vdash (\mathsf{t}_2 \ll |\tau_2|\ \mathsf{t'}_2) \Downarrow (\rho_2,\dot{v}_2) \quad \exists i \in [1,2], \dot{v}_i = \mathsf{reject} \quad \mathit{symb}(\mathsf{t}) \in \mathcal{F}^\star \vee \mathit{symb}(\mathsf{t'}) \in \mathcal{F}^\star}{\rho \Vdash \mathsf{t} \ll |\tau|\ \mathsf{t'} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MListRej}$$

Their premises depend on the application of one of small-step semantics rules presented in Chapter 3 of Radu Kopetz's thesis [Kop08]: Mutate, $\mathsf{SymbolClash}_1^+$ or $\mathsf{SymbolClash}_2^+$. Each one of these rules have two premises involving terms t_i and $\mathsf{t'}_i$ composed not only of terms t and t' in conclusion but of fresh terms. Likewise, the rules [E-MAntiAcc] and [E-MAntiRej] for the evaluation of an anti-matching condition (i.e. a matching condition containing anti-patterns) correspond to the small-step semantics rule ElimAnti defined by Radu Kopetz:

$$\frac{\mathsf{x} \in \mathcal{V} \quad \tau =_t \mathsf{typeof}([\,],\mathsf{x}) \quad \tau =_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x} \ll |\tau|\ \dot{u} \Downarrow (\{\rho \lhd \{\mathsf{x} \mapsto \dot{u}\}\}, \mathsf{accept})}\ \text{E-MVarAcc}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad \tau \neq_t \mathsf{typeof}([\,],\mathsf{x}) \vee \tau \neq_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x} \ll |\tau|\ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MVarRej}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad symb(\dot{u}) \in \mathcal{F}^\star \quad \tau =_t \mathsf{typeof}([\,],\mathsf{x}) \quad \tau =_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x}^* \ll |\tau|\ \dot{u} \Downarrow (\{\rho \lhd \{\mathsf{x} \mapsto \dot{u}\}\}, \mathsf{accept})}\ \text{E-MSVarAcc}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad symb(\dot{u}) \in \mathcal{F}^\star \quad \tau \neq_t \mathsf{typeof}([\,],\mathsf{x}) \vee \tau \neq_t \mathsf{typeof}([\,],\dot{u})}{\rho \Vdash \mathsf{x}^* \ll |\tau|\ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MSVarRej}$$

$$\frac{\mathsf{x} \in \mathcal{V} \quad \rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\rho', \dot{v})}{\rho \Vdash \mathsf{x@t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho' \lhd \{\mathsf{x} \mapsto \dot{u}\}\}, \dot{v})}\ \text{E-MAlias}$$

$$\frac{\begin{array}{c}\rho_0 \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \dot{u}_1) \Downarrow (\rho_1, \mathsf{accept}) \quad \ldots \quad \rho_0 \Vdash (\mathsf{t}_n \ll |\tau_n|\ \dot{u}_n) \Downarrow (\rho_n, \mathsf{accept}) \\ \mathsf{f} \in \mathcal{F} \quad \tau =_t \mathsf{typeof}([\,],\mathsf{f}(\dot{u}_1, \ldots, \dot{u}_n)) \quad \forall i \in [1,n], \rho_{i-1} \bowtie \rho_i\end{array}}{\rho_0 \Vdash \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \ll |\tau|\ \mathsf{f}(\dot{u}_1, \ldots, \dot{u}_n) \Downarrow (\{\bigcup_{j=0}^{n} \rho_j\}, \mathsf{accept})}\ \text{E-MFunAcc}$$

$$\frac{\begin{array}{c}\rho_0 \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \dot{u}_1) \Downarrow (\rho_1, \dot{v}_1) \quad \ldots \quad \rho_0 \Vdash (\mathsf{t}_n \ll |\tau_n|\ \dot{u}_n) \Downarrow (\rho_n, \dot{v}_n) \\ \mathsf{f} \in \mathcal{F} \quad \exists i \in [1,n], (\dot{v}_i = \mathsf{reject} \vee \rho_{i-1} \not\bowtie \rho_i)\end{array}}{\rho_0 \Vdash \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \ll |\tau|\ \mathsf{f}(\dot{u}_1, \ldots, \dot{u}_n) \Downarrow (\{\rho_0\}, \mathsf{reject})}\ \text{E-MFunRej}$$

$$\frac{\mathsf{g}, \mathsf{g'} \in \mathcal{F} \cup \mathcal{F}^\star \quad \mathsf{g} \neq \mathsf{g'} \vee \tau \neq_t \mathsf{typeof}([\,],\mathsf{g'}(\dot{u}_1, \ldots, \dot{u}_n))}{\rho \Vdash \mathsf{g}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \ll |\tau|\ \mathsf{g'}(\dot{u}_1, \ldots, \dot{u}_n) \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MFunListRej}$$

$$\frac{\rho \Vdash \mathsf{t'} \Downarrow (\rho, \dot{u}) \quad \rho \Vdash (\mathsf{t} \ll |\tau|\ \dot{u})) \Downarrow (\rho', \dot{v}) \quad \mathcal{V}ar(\mathsf{t'}) \neq \emptyset}{\rho \Vdash (\mathsf{t} \ll |\tau|\ \mathsf{t'}) \Downarrow (\{\rho'\}, \dot{v})}\ \text{E-MSubj}$$

where $\mathsf{t}, \mathsf{t}_i, \mathsf{t'}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ and $\dot{u}, \dot{u}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ for $i \in [1,n]$.

Figure 3.2: Evaluation rules for syntatic matching conditions.

$$\frac{\rho \Vdash \mathsf{t}[\mathsf{x}]_\omega \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \mathsf{accept}) \quad \rho' \Vdash \mathit{not}(\mathsf{t}[\mathsf{t}_1]_\omega \ll |\tau|\ \dot{u}) \Downarrow (\{\rho'\}, \mathsf{accept}) \quad \mathsf{x} \in \mathcal{V}\backslash\mathcal{V}ar(\mathsf{t}) \quad \mathsf{t}_{|\omega} = !\mathsf{t}_1}{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \mathsf{accept})}\ \text{E-MAntiAcc}$$

and

$$\frac{\rho \Vdash \mathsf{t}[\mathsf{x}]_\omega \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \dot{v}_1) \quad \rho' \Vdash \mathit{not}(\mathsf{t}[\mathsf{t}_1]_\omega \ll |\tau|\ \dot{u}) \Downarrow (\{\rho'\}, \dot{v}_2) \quad \mathsf{x} \in \mathcal{V}\backslash\mathcal{V}ar(\mathsf{t}) \quad \mathsf{t}_{|\omega} = !\mathsf{t}_1 \quad \exists i \in [1,2], \dot{v}_i = \mathsf{reject}}{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MAntiRej}$$

The subterm *not* occurring in their premises does not belong to the Tom core abstract syntax (see Definition 3.5) and its evaluation is performed by the $\mathcal{AU}$-AntiMatch algorithm also defined in Radu Kopetz's thesis. Here, we simulate the evaluation of a term with *not* as head symbol by two auxiliary evaluation rules [E-NotAcc] and [E-NotRej]:

$$\frac{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \dot{v}) \quad (\dot{v} = \mathsf{reject} \vee \rho \not\bowtie \rho')}{\rho \Vdash \mathit{not}(\mathsf{t} \ll |\tau|\ \dot{u}) \Downarrow (\{\rho\}, \mathsf{accept})}\ \text{E-NotAcc}$$

and

$$\frac{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \dot{v}) \quad \dot{v} = \mathsf{accept} \quad \rho \bowtie \rho'}{\rho \Vdash \mathit{not}(\mathsf{t} \ll |\tau|\ \dot{u}) \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-NotRej}$$

The $\mathcal{AU}$-AntiMatch algorithm also gives a higher priority to rules [E-MAntiAcc] and [E-MAntiRej]. Thus, an anti-matching condition $\mathsf{f}(!\mathsf{x}) \ll |\tau|\ \dot{u}$ must be first evaluated by either [E-MAntiAcc] or [E-MAntiRej] and then the other rules such as [E-MFunAcc] and [E-MFunRej] are applied. The rules for associative matching and anti-pattern matching are presented in Figure 3.3.

For numeric conditions, the rules [E-NumAcc] and [E-NumRej] apply evaluating both terms compared by the operator $\diamond$ and producing a singleton containing the unaltered environment ρ:

$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \rho \Vdash \mathsf{t}_2 \Downarrow \dot{u}_2 \quad \dot{u}_1 \diamond \dot{u}_2}{\rho \Vdash (\mathsf{t}_1 \diamond\ \mathsf{t}_2) \Downarrow (\{\rho\}, \mathsf{accept})}\ \text{E-NumAcc}$$

and

$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \rho \Vdash \mathsf{t}_2 \Downarrow \dot{u}_2 \quad \neg(\dot{u}_1 \diamond \dot{u}_2)}{\rho \Vdash (\mathsf{t}_1 \diamond\ \mathsf{t}_2) \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-NumRej}$$

Conditions composed of (nested) conjunctions and/or disjunctions are consid-

$$\frac{\rho \Vdash \mathsf{t}[\mathsf{x}]_\omega \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \mathsf{accept}) \quad \rho' \Vdash not(\mathsf{t}[\mathsf{t}_1]_\omega \ll |\tau|\ \dot{u}) \Downarrow (\{\rho'\}, \mathsf{accept}) \quad \mathsf{x} \in \mathcal{V}\backslash\mathcal{V}ar(\mathsf{t}) \quad \mathsf{t}_{|\omega} = !\mathsf{t}_1}{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \mathsf{accept})}\ \text{E-MAntiAcc}$$

$$\frac{\rho \Vdash \mathsf{t}[\mathsf{x}]_\omega \ll |\tau|\ \dot{u} \Downarrow (\{\rho'\}, \dot{v}_1) \quad \rho' \Vdash not(\mathsf{t}[\mathsf{t}_1]_\omega \ll |\tau|\ \dot{u}) \Downarrow (\{\rho'\}, \dot{v}_2) \quad \mathsf{x} \in \mathcal{V}\backslash\mathcal{V}ar(\mathsf{t}) \quad \mathsf{t}_{|\omega} = !\mathsf{t}_1 \quad \exists i \in [1,2], \dot{v}_i = \mathsf{reject}}{\rho \Vdash \mathsf{t} \ll |\tau|\ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MAntiRej}$$

$$\frac{\rho \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \mathsf{t'}_1) \Downarrow (\rho_1, \mathsf{accept}) \quad \rho_1 \Vdash (\mathsf{t}_2 \ll |\tau_2|\ \mathsf{t'}_2) \Downarrow (\rho_2, \mathsf{accept}) \quad \rho \bowtie \rho_1 \quad \rho_1 \bowtie \rho_2 \quad symb(\mathsf{t}) \in \mathcal{F}^\star \vee symb(\mathsf{t'}) \in \mathcal{F}^\star}{\rho \Vdash \mathsf{t} \ll |\tau|\ \mathsf{t'} \Downarrow (\{\rho_2\}, \mathsf{accept})}\ \text{E-MListAcc}$$

$$\frac{\rho \Vdash (\mathsf{t}_1 \ll |\tau_1|\ \mathsf{t'}_1) \Downarrow (\rho_1, \dot{v}_1) \quad \rho_1 \Vdash (\mathsf{t}_2 \ll |\tau_2|\ \mathsf{t'}_2) \Downarrow (\rho_2, \dot{v}_2) \quad \exists i \in [1,2], \dot{v}_i = \mathsf{reject} \quad symb(\mathsf{t}) \in \mathcal{F}^\star \vee symb(\mathsf{t'}) \in \mathcal{F}^\star}{\rho \Vdash \mathsf{t} \ll |\tau|\ \mathsf{t'} \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-MListRej}$$

where $\mathsf{t}, \mathsf{t}_i, \mathsf{t'}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ and $\dot{u}, \dot{u}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$ for $i \in [1, n]$.

Figure 3.3: Evaluation rules for associative (anti-pattern) matching conditions.

ered in disjunctive normal form. Each condition of a conjunction is evaluated by either the rule [E-ConjAcc] or the rule [E-ConjRej]:

$$\frac{\rho \Vdash cond_1 \Downarrow (\{\rho_1\}, \mathsf{accept}) \quad \rho_1 \Vdash cond_2 \Downarrow (\{\rho_2\}, \mathsf{accept})}{\rho \Vdash cond_1 \wedge cond_2 \Downarrow (\{\rho_2\}, \mathsf{accept})}\ \text{E-ConjAcc}$$

or

$$\frac{\rho \Vdash cond_1 \Downarrow (\{\rho_1\}, \dot{v}_1) \quad \rho_1 \Vdash cond_2 \Downarrow (\{\rho_2\}, \dot{v}_2) \quad \exists i \in [1,2], \dot{v}_i = \mathsf{reject}}{\rho \Vdash cond_1 \wedge cond_2 \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-ConjRej}$$

The evaluation of a sequence of actions occurring in the right-hand side of a rule always evaluates to accept by the following rule:

$$\frac{\rho \Vdash action_1 \Downarrow (\rho_1, \dot{v}_1) \quad \ldots \quad \rho_{n-1} \Vdash action_n \Downarrow (\rho_n, \dot{v}_n)}{\rho \Vdash (action_1; \ldots; action_n) \Downarrow (\rho_n, \mathsf{accept})}\ \text{E-Action}$$

The result of application of [E-Action] is independent of the evaluation of actions since they are all possible but not necessarily executed. The evaluation of a disjunction of conditions in left-hand side of rules is described by rules

[E-ExDisjAcc], [E-DisjAcc] and [E-DisjRej] which may produce a new set of environments:

$$\frac{\rho \Vdash cond_1 \Downarrow (\{\rho_{1,1}, \ldots, \rho_{1,n_1}\}, \dot{v}_1) \quad \rho \Vdash cond_2 \Downarrow (\{\rho_{2,1}, \ldots, \rho_{2,n_2}\}, \dot{v}_2) \quad \exists i, j \in [1,2], (\dot{v}_i = \mathsf{accept} \wedge \dot{v}_j = \mathsf{reject} \wedge i \neq j)}{\rho \Vdash cond_1 \vee cond_2 \Downarrow (\{\rho_{i,1}, \ldots, \rho_{i,n_i}\}, \mathsf{accept})} \text{ E-ExDisjAcc}$$

and

$$\frac{\rho \Vdash cond_1 \Downarrow (\{\rho_1, \ldots, \rho_n\}, \mathsf{accept}) \quad \rho \Vdash cond_2 \Downarrow (\{\rho'_1, \ldots, \rho'_m\}, \mathsf{accept})}{\rho \Vdash cond_1 \vee cond_2 \Downarrow (\{\rho_1, \ldots, \rho_n\} \cup \{\rho'_1, \ldots, \rho'_m\}, \mathsf{accept})} \text{ E-DisjAcc}$$

and

$$\frac{\rho \Vdash cond_1 \Downarrow (\{\rho_1, \ldots, \rho_n\}, \mathsf{reject}) \quad \rho \Vdash cond_2 \Downarrow (\{\rho'_1, \ldots, \rho'_m\}, \mathsf{reject})}{\rho \Vdash cond_1 \vee cond_2 \Downarrow (\{\rho\}, \mathsf{reject})} \text{ E-DisjRej}$$

This happens because the evaluation of a rule $cond_1 \vee cond_2 \longrightarrow action$ is semantically equivalent to the evaluation of a block with two Tom rules $\{cond_1 \longrightarrow action; cond_2 \longrightarrow action\}$ in two different environments. Blocks are handled by the rule [E-Rule] which describes the evaluation of a (Tom) rule depending on the evaluation predominantly of the left-hand side and occasionally of the right-hand side:

$$\frac{\rho \Vdash cond \Downarrow (\{\rho_1, \ldots, \rho_n\}, \dot{v}) \quad \rho_1 \Vdash action \Downarrow (\rho'_1, \mathsf{accept}) \quad \ldots \quad \rho_n \Vdash action \Downarrow (\rho'_n, \mathsf{accept})}{\rho \Vdash cond \longrightarrow action \Downarrow (\rho, \dot{v})} \text{ E-Rule}$$

The semantics of a block of Tom rules is different from most of functional languages: when the left-hand side of a rule is accepted, the action of the right-hand side is evaluated and then the next rule is evaluated, unless the execution flow is interrupted. This is described respectively by the rules [E-BlockAcc] and [E-BlockRej] which result in accept, if at least one Tom rule is accepted, or in reject, otherwise:

$$\frac{\rho \Vdash rule_1 \Downarrow (\rho, \dot{v}_1) \quad \ldots \quad \rho \Vdash rule_n \Downarrow (\rho, \dot{v}_n) \quad \exists i, \dot{v}_i = \mathsf{accept}}{\rho \Vdash \{rule_1; \ldots; rule_n\} \Downarrow (\rho, \mathsf{accept})} \text{ E-BlockAcc}$$

and

$$\frac{\rho \Vdash rule_1 \Downarrow (\rho, \dot{v}_1) \quad \ldots \quad \rho \Vdash rule_n \Downarrow (\rho, \dot{v}_n) \quad \forall i, \dot{v}_i = \mathsf{reject}}{\rho \Vdash \{rule_1; \ldots; rule_n\} \Downarrow (\rho, \mathsf{reject})} \text{ E-BlockRej}$$

All these semantics rules describing the evaluation of expressions which are neither terms nor matching conditions are summarized in Figure 3.4.

Example 3.12. *Considering Example 3.6, one of the possible evaluations for the expression*

$$\{\mathsf{concNat(x*,pnat@suc(y),z*)} \ll |\mathit{NatList}^{?}|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}$$

is illustrated in Figures 3.5 and 3.6.

$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \rho \Vdash \mathsf{t}_2 \Downarrow \dot{u}_2 \quad \dot{u}_1 \diamond \dot{u}_2}{\rho \Vdash (\mathsf{t}_1 \diamond \mathsf{t}_2) \Downarrow (\{\rho\}, \mathsf{accept})}\ \text{E-NumAcc}$$

$$\frac{\rho \Vdash \mathsf{t}_1 \Downarrow \dot{u}_1 \quad \rho \Vdash \mathsf{t}_2 \Downarrow \dot{u}_2 \quad \neg(\dot{u}_1 \diamond \dot{u}_2)}{\rho \Vdash (\mathsf{t}_1 \diamond \mathsf{t}_2) \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-NumRej}$$

$$\frac{\rho \Vdash cond_1 \Downarrow (\{\rho_1\}, \mathsf{accept}) \quad \rho_1 \Vdash cond_2 \Downarrow (\{\rho_2\}, \mathsf{accept})}{\rho \Vdash cond_1 \wedge cond_2 \Downarrow (\{\rho_2\}, \mathsf{accept})}\ \text{E-ConjAcc}$$

$$\frac{\begin{array}{c}\rho \Vdash cond_1 \Downarrow (\{\rho_1\}, \dot{v}_1) \quad \rho_1 \Vdash cond_2 \Downarrow (\{\rho_2\}, \dot{v}_2) \\ \exists i \in [1,2], \dot{v}_i = \mathsf{reject}\end{array}}{\rho \Vdash cond_1 \wedge cond_2 \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-ConjRej}$$

$$\frac{\begin{array}{c}\rho \Vdash cond_1 \Downarrow (\{\rho_{1,1}, \ldots, \rho_{1,n_1}\}, \dot{v}_1) \quad \rho \Vdash cond_2 \Downarrow (\{\rho_{2,1}, \ldots, \rho_{2,n_2}\}, \dot{v}_2) \\ \exists i, j \in [1,2], (\dot{v}_i = \mathsf{accept} \wedge \dot{v}_j = \mathsf{reject} \wedge i \neq j)\end{array}}{\rho \Vdash cond_1 \vee cond_2 \Downarrow (\{\rho_{i,1}, \ldots, \rho_{i,n_i}\}, \mathsf{accept})}\ \text{E-ExDisjAcc}$$

$$\frac{\rho \Vdash cond_1 \Downarrow (\{\rho_1, \ldots, \rho_n\}, \mathsf{accept}) \quad \rho \Vdash cond_2 \Downarrow (\{\rho'_1, \ldots, \rho'_m\}, \mathsf{accept})}{\rho \Vdash cond_1 \vee cond_2 \Downarrow (\{\rho_1, \ldots, \rho_n\} \cup \{\rho'_1, \ldots, \rho'_m\}, \mathsf{accept})}\ \text{E-DisjAcc}$$

$$\frac{\rho \Vdash cond_1 \Downarrow (\{\rho_1, \ldots, \rho_n\}, \mathsf{reject}) \quad \rho \Vdash cond_2 \Downarrow (\{\rho'_1, \ldots, \rho'_m\}, \mathsf{reject})}{\rho \Vdash cond_1 \vee cond_2 \Downarrow (\{\rho\}, \mathsf{reject})}\ \text{E-DisjRej}$$

$$\frac{\rho \Vdash action_1 \Downarrow (\rho_1, \dot{v}_1) \quad \ldots \quad \rho_{n-1} \Vdash action_n \Downarrow (\rho_n, \dot{v}_n)}{\rho \Vdash (action_1; \ldots; action_n) \Downarrow (\rho_n, \mathsf{accept})}\ \text{E-Action}$$

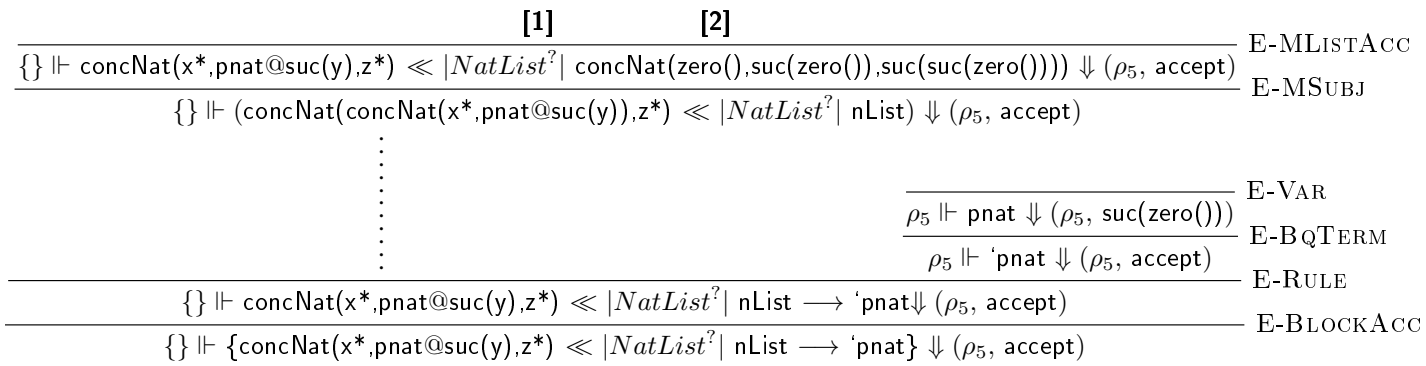
$$\frac{\begin{array}{c}\rho \Vdash cond \Downarrow (\{\rho_1, \ldots, \rho_n\}, \dot{v}) \\ \rho_1 \Vdash action \Downarrow (\rho'_1, \mathsf{accept}) \quad \ldots \quad \rho_n \Vdash action \Downarrow (\rho'_n, \mathsf{accept})\end{array}}{\rho \Vdash cond \longrightarrow action \Downarrow (\rho, \dot{v})}\ \text{E-Rule}$$

$$\frac{\rho \Vdash rule_1 \Downarrow (\rho, \dot{v}_1) \quad \ldots \quad \rho \Vdash rule_n \Downarrow (\rho, \dot{v}_n) \quad \exists i, \dot{v}_i = \mathsf{accept}}{\rho \Vdash \{rule_1; \ldots; rule_n\} \Downarrow (\rho, \mathsf{accept})}\ \text{E-BlockAcc}$$

$$\frac{\rho \Vdash rule_1 \Downarrow (\rho, \dot{v}_1) \quad \ldots \quad \rho \Vdash rule_n \Downarrow (\rho, \dot{v}_n) \quad \forall i, \dot{v}_i = \mathsf{reject}}{\rho \Vdash \{rule_1; \ldots; rule_n\} \Downarrow (\rho, \mathsf{reject})}\ \text{E-BlockRej}$$

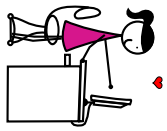
where $\mathsf{t}_1, \mathsf{t}_2 \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ and $\dot{u}_1, \dot{u}_2 \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$.

Figure 3.4: Evaluation rules for numeric conditions and remaining expressions.



$\rho_1 = \{\mathsf{w} \mapsto \mathsf{suc(suc(zero()))}\}$
$\rho_2 = \{\mathsf{w} \mapsto \mathsf{suc(suc(zero()))},\ \mathsf{z} \mapsto \mathsf{concNat()}\}$
$\rho_3 = \{\mathsf{w} \mapsto \mathsf{suc(suc(zero()))},\ \mathsf{z} \mapsto \mathsf{concNat()},\ \mathsf{x} \mapsto \mathsf{concNat(zero(),suc(zero()))}\}$
$\rho_4 = \{\mathsf{w} \mapsto \mathsf{suc(suc(zero()))},\ \mathsf{z} \mapsto \mathsf{concNat()},\ \mathsf{x} \mapsto \mathsf{concNat(zero(),suc(zero()))},\ \mathsf{y} \mapsto \mathsf{suc(zero())}\}$
$\rho_5 = \{\mathsf{w} \mapsto \mathsf{suc(suc(zero()))},\ \mathsf{z} \mapsto \mathsf{concNat()},\ \mathsf{x} \mapsto \mathsf{concNat(zero(),suc(zero()))},\ \mathsf{y} \mapsto \mathsf{suc(zero())},\ \mathsf{pNat} \mapsto \mathsf{suc(suc(zero()))}\}$

Figure 3.5: Example using the big-step semantics of Tom.



$$\dfrac{\dfrac{\dfrac{\dfrac{}{\rho_3 \Vdash \mathsf{y} \ll |Nat^?| \ \mathsf{suc(zero())} \Downarrow (\rho_4,\ \mathsf{accept})}\ \text{E-MVarAcc}}{\rho_3 \Vdash \mathsf{suc(y)} \ll |Nat^?| \ \mathsf{suc(suc(zero()))} \Downarrow (\rho_4,\ \mathsf{accept})}\ \text{E-MFunAcc}}{\rho_3 \Vdash \mathsf{pnat@suc(y)} \ll |Nat^?| \ \mathsf{suc(suc(zero()))} \Downarrow (\rho_5,\ \mathsf{accept})}\ \text{E-MAlias}}{\rho_2 \Vdash \mathsf{x^*} \ll |NatList^?| \ \mathsf{concNat(zero(),suc(zero()))} \Downarrow (\rho_3,\ \mathsf{accept})}\ \text{E-MSVarAcc}$$

$$\vdots$$

$$\dfrac{\rho_2 \Vdash \mathsf{x^*} \ll |NatList^?| \ \mathsf{concNat(zero(),suc(zero()))} \Downarrow (\rho_3,\ \mathsf{accept})}{\rho_2 \Vdash \mathsf{concNat(x^*,pnat@suc(y))} \ll |NatList^?| \ \mathsf{concNat(concNat(zero(),suc(zero())),suc(suc(zero())))} \Downarrow (\rho_5,\ \mathsf{accept})}\ \text{E-MListAcc}$$

$$\vdots$$

$$\dfrac{\dfrac{}{\rho_2 \Vdash \mathsf{w} \Downarrow (\rho_2,\ \mathsf{suc(suc(zero()))})}\ \text{E-Var}}{\rho_2 \Vdash \mathsf{concNat(x^*,pnat@suc(y))} \ll |NatList^?| \ \mathsf{concNat(concNat(zero(),suc(zero())),w)} \Downarrow (\rho_5,\ \mathsf{accept})}\ \text{E-MSubj}$$

[2]

$$\dfrac{\dfrac{}{\{\} \Vdash \mathsf{w} \ll |Nat^?| \ \mathsf{suc(suc(zero()))} \Downarrow (\rho_1,\ \mathsf{accept})}\ \text{E-MVarAcc} \quad \dfrac{}{\rho_1 \Vdash \mathsf{z^*} \ll |NatList^?| \ \mathsf{concNat()} \Downarrow (\rho_2,\ \mathsf{accept})}\ \text{E-MSVarAcc}}{\{\} \Vdash \mathsf{concNat(w,z^*)} \ll |NatList^?| \ \mathsf{suc(suc(zero()))} \Downarrow (\rho_2,\ \mathsf{accept})}\ \text{E-MListAcc}$$

[1]

Figure 3.6: (continuation of Figure 3.5) Example using the big-step semantics of Tom.

3.3.3 Type checking

Type and data constructors correspond respectively to base sorts and (syntactic and variadic) operators. Tom programs are seen as programs with explicit type annotation which enables the evaluation of type information of Tom expressions to be decidable at compile time by the use of static typing. Accordingly, stating that "a term t is of type τ" means that we can conclude statically that t evaluates to a value of type τ. In order to eliminate Tom programs which cause errors at runtime, we describe a type checking system which verifies automatically that such programs are well-typed.

Preliminary definitions

The typing rules of our type checking system deal with a *context* defined as a set of pairs (variable,type) and (operator,rank).

Definition 3.13 (Context). *A* context *is defined by:*

$$\Gamma ::= \varnothing \mid \Gamma; \mathsf{x} : \tau \mid \Gamma; \mathsf{f} : s_1^?, \ldots, s_n^? \to s^? \mid \Gamma; \mathsf{v} : (s_1^?)^\star \to s^v$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$, $\mathsf{v} \in \mathcal{F}^\star$, $\tau \in \mathcal{T}y$ *and* $s^?, s_i^?, s^v \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\})$. *Moreover, variadic operators* v *having an indefinite arity with domain* $(s_1^?)^\star$ *and codomain* s^v *are written* $\mathsf{v} : (s_1^?)^\star \to s^v$, *according to Definition 1.25.*

A context Γ associates types to variables. It also associates ranks to either syntactic or variadic operators. We denote by $\Gamma(\gamma)$ the fact that a pair γ belongs to Γ. The context has at most one declaration of type per variable and one declaration of rank per operator since operator overloading is forbidden. The initialization of the context occurs during parsing: each variable of a (matching or numeric) condition is paired with its sort decorated with ? while each algebraic operator of a Tom mapping is paired with its rank. The ranks of algebraic operators are built from decoration of their sorts in the following way:

- for a syntactic operator f: all sorts appearing in the rank are decorated with ?,
- for a variadic operator v: all sorts appearing in the domain are decorated with ? while the sort of the codomain is decorated with v.

In fact, Tom ignores Java code, but for the purpose of verifying types, we consider that the rank of Java operators and types of Java variables are known. Consequently, Java operators (represented by Tom operators) and their ranks as well as Java variables and their types are also declared into the context. Furthermore, context access is defined by the function typeof(Γ,t) which returns the type of the term t in the context Γ.

Definition 3.14 (Context access)**.** *The* context access *is done by a partial binary function* $\mathsf{typeof} : \Gamma \times \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V}) \to \mathcal{T}y$ *defined by:*

$\mathsf{typeof}(\Gamma,\mathsf{x}) = \tau$, *if* $\mathsf{x} : \tau \in \Gamma$
$\mathsf{typeof}(\Gamma,\mathsf{x@t}) = \mathsf{typeof}(\Gamma,\mathsf{t})$
$\mathsf{typeof}(\Gamma,\mathsf{!t}) = \mathsf{typeof}(\Gamma,\mathsf{t})$
$\mathsf{typeof}(\Gamma,\mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) = s^?$, *if* $\mathsf{f} : s_1^?,\ldots,s_n^? \to s^? \in \Gamma$
$\mathsf{typeof}(\Gamma,\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) = s^v$, *if* $\mathsf{v} : (s_1^?)^\star \to s^v \in \Gamma$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$, $\mathsf{v} \in \mathcal{F}^\star$, $s^?, s_i^?, s^v \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\})$ *and* $\mathsf{t},\mathsf{t}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *for* $i \in [1,n]$.

Type checking rules

A typing judgment for Tom code has the form $\Gamma \vdash e : \tau$ and is defined by a set of inference rules assigning types to Tom expressions, summarized in Figure 3.7. We recall that wt is a type which denotes the well-typedness of those Tom expressions which are neither Tom terms nor patterns. The typing rules are standard except for the use of decorated sort as types. All variables of a typing rule are to be considered as universally quantified. Moreover, we consider the variables occurring in patterns of different *rule* expressions modulo renaming. Starting from a context Γ and a Tom expression e, we say that e is *well-typed* if it is possible to build a typing derivation to it.

Definition 3.15 (Well-typed expression and code)**.** *A Tom expression* e *is* well-typed *in a context* Γ *if there exists a type* $\tau \in \mathcal{T}y \backslash \mathcal{X}$ *such that* $\Gamma \vdash e : \tau$.

A Tom code is well-typed *in a context* Γ *if and only if all its expressions are well-typed.*

In Figure 3.7, the rule [T-Var] is standard: variable x is of type s^h as expected for it to be in Γ. The rule [T-SVar] is similar but applying to star variables. Anti-patterns and alias are introduced respectively by rules [T-Anti] and [T-Alias]. In the rule [T-Fun], assuming that $s_1^?,\ldots,s_n^? \to s^?$ is the rank of f in Γ, if each term t_i, $i \in [1,n]$ is of type $s_i^?$ then the application of f to $\mathsf{t}_1,\ldots,\mathsf{t}_n$ is of type $s^?$. We note the presence of the rule [Gen] which makes $\Gamma \vdash \mathsf{t} : s^?$ a valid judgment if t can be typed by $\Gamma \vdash \mathsf{t} : s^v$.

The most interesting rules are those that apply to lists. There are three: [T-Empty] checks if an empty list is of the same type declared in Γ; [T-Elem] is similar to [T-Fun] but is applied to variadic operators; and [T-Merge] is applied to a concatenation of two lists of type s^v in Γ, resulting in a new list of same type s^v. The application conditions of rules [T-Elem] and [T-Merge] ensure the determinism of type checking: this property is useful to prove Theorem **??**. For assignment instructions, rule [T-Ass] ensures that both variable and value introduced by a backquoted term must be of a same type. Backquoted terms in their turn are either (Tom) algebraic terms or Java objects whose typing is introduced by [T-BqTerm].

The remaining rules only build expressions without verification of types of terms, except for rules [T-Match] and [T-Num] which introduce a (matching or numeric) well-typed condition if both terms involved in it are of a same type $s^?$.

$$\frac{}{\Gamma(\mathsf{x} : s^h) \vdash \mathsf{x} : s^h}\ \text{T-Var} \qquad \frac{}{\Gamma(\mathsf{x} : s^v) \vdash \mathsf{x*} : s^v}\ \text{T-SVar}$$

$$\frac{\Gamma \vdash \mathsf{t} : s^v}{\Gamma \vdash \mathsf{t} : s^?}\ \text{Gen} \qquad \frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma(\mathsf{x} : s^h) \vdash \mathsf{x@t} : s^h}\ \text{T-Alias}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : s_1^? \quad \ldots \quad \Gamma \vdash \mathsf{t}_n : s_n^?}{\Gamma(\mathsf{f} : s_1^?, \ldots, s_n^? \to s^?) \vdash \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^?}\ \text{T-Fun} \qquad \frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma \vdash \mathsf{!t} : s^h}\ \text{T-Anti}$$

$$\frac{}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \mathsf{v}() : s^v}\ \text{T-Empty}$$

$$\frac{\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v \qquad \Gamma \vdash \mathsf{t}_n : s_1^?}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v}\ \text{T-Elem}$$
$$\text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v \text{ and } \mathsf{t}_n \neq \mathsf{x*}$$

$$\frac{\Gamma \vdash \mathsf{v}(\mathsf{t}_1, \ldots,\mathsf{t}_{n-1}) : s^v \qquad \Gamma \vdash \mathsf{t}_n : s^v}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \mathsf{v}(\mathsf{t}_1, \ldots,\mathsf{t}_n) : s^v}\ \text{T-Merge}$$
$$\text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v \text{ or } \mathsf{t}_n = \mathsf{x*}$$

$$\frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma(\mathsf{x} : s^h) \vdash \mathsf{x} := \text{`}\mathsf{t} : wt}\ \text{T-Ass} \qquad \frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma \vdash \text{`}\mathsf{t} : wt}\ \text{T-BqTerm}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : s^? \qquad \Gamma \vdash \mathsf{t}_2 : s^?}{\Gamma \vdash (\mathsf{t}_1 \ll |s^h|\ \mathsf{t}_2) : wt}\ \text{T-Match} \qquad \frac{\Gamma \vdash \mathsf{t}_1 : s^? \qquad \Gamma \vdash \mathsf{t}_2 : s^?}{\Gamma \vdash (\mathsf{t}_1 \diamond\ \mathsf{t}_2) : wt}\ \text{T-Num}$$
$$\text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_i) = s^{h_i} \text{ for } i \in [1,2]$$

$$\frac{\Gamma \vdash (cond_1) : wt \qquad \Gamma \vdash (cond_2) : wt}{\Gamma \vdash (cond_1 \wedge cond_2) : wt}\ \text{T-Conj}$$

$$\frac{\Gamma \vdash (cond_1) : wt \qquad \Gamma \vdash (cond_2) : wt}{\Gamma \vdash (cond_1 \vee cond_2) : wt}\ \text{T-Disj}$$

$$\frac{\Gamma \vdash action_1 : wt \qquad \ldots \qquad \Gamma \vdash action_n : wt}{\Gamma \vdash (action_1; \ldots; action_n) : wt}\ \text{T-Action}$$

$$\frac{\Gamma \vdash cond : wt \qquad \Gamma \vdash action : wt}{\Gamma \vdash (cond \longrightarrow action) : wt}\ \text{T-Rule}$$

$$\frac{\Gamma \vdash rule_1 : wt \qquad \ldots \qquad \Gamma \vdash rule_n : wt}{\Gamma \vdash \{rule_1; \ldots; rule_n\} : wt}\ \text{T-Block}$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$, $\mathsf{v} \in \mathcal{F}^\star$ and $h \in \mathcal{F}^\star \cup \{?\}$

Figure 3.7: Type checking rules for (backquoted) terms and expressions.

The majority of type checking rules are syntax directed except for those having an applicability condition which uses the function typeof. In fact, they define an algorithm of type checking that reads derivations bottom-up. In practice, the rule [Gen] is combined with other rules and the algorithm stops when none of the type checking rules can be applied. The original Tom expression is well-typed if each branch of derivation reaches the [T-Var], [T-SVar] or [T-Empty] cases or even the rule [T-Fun] applied to a constant. Otherwise, the algorithm raises an error.

Example 3.16. *Let* $\Gamma = (\mathsf{suc} : Nat^? \to Nat^?; \mathsf{concNat} : (Nat^?)^\star \to NatList^{concNat}; \mathsf{x} : NatList^{concNat}; \mathsf{y} : Nat^?; \mathsf{z} : NatList^{concNat}; \mathsf{pnat} : Nat^?; \mathsf{nList} : NatList^{concNat})$. *Considering Example 3.6, the expression*

$$\{\mathsf{concNat(x*,pnat@suc(y),z*)} \ll |NatList^?|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}$$

is well-typed and its deduction tree is given in Figure 3.8

Properties of type checking

In this section, we prove the safety of our type system, i.e. that a Tom program composed of well-typed code cannot cause errors at runtime. We start by introducing an auxiliary lemma useful to reason about typing judgments considering the shape of type checking rules.

Lemma 3.17 (Inversion of the typing judgment)**.** *There exists a typing derivation for each valid typing judgments according to the shape of the applicable type checking rules. Let* x *be a variable and* f, v *be operators such that* $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$ *and* $\mathsf{v} \in \mathcal{F}^\star$. *Let* $\tau \in \mathcal{T}y \backslash \mathcal{X}$. *Let* $h, h_i \in \mathcal{F}^\star \cup \{?\}$ *and* t, $\mathsf{t}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *for* $i \in [1, n]$.

1. *If* $\Gamma \vdash \mathsf{x} : \tau$, *then:*
   (a) *either there is a type* s^h *such that* $\tau = s^h$ *with* $\mathsf{x} : s^h \in \Gamma$,
   (b) *or there is a type* $s^?$ *such that* $\tau = s^?$ *with* $\Gamma \vdash \mathsf{x} : s^v$ *for some* $\mathsf{v} \in \mathcal{F}^\star$.
2. *If* $\Gamma \vdash \mathsf{x*} : \tau$, *then:*
   (a) *either there is a type* s^v *such that* $\tau = s^v$ *with* $\mathsf{x} : s^v \in \Gamma$,
   (b) *or there is a type* $s^?$ *such that* $\tau = s^?$ *with* $\Gamma \vdash \mathsf{x*} : s^v$ *for some* $\mathsf{v} \in \mathcal{F}^\star$.
3. *If* $\Gamma \vdash \mathsf{!t} : \tau$, *then there is a type* s^h *such that* $\tau = s^h$ *with* $\Gamma \vdash \mathsf{t} : s^h$.
4. *If* $\Gamma \vdash \mathsf{x@t} : \tau$, *then there is a type* s^h *such that* $\mathsf{x} : s^h \in \Gamma$ *and* $\Gamma \vdash \mathsf{t} : s^h$.
5. *If* $\Gamma \vdash \mathsf{f(t_1,\ldots,t_n)} : \tau$, *then there are some types* $s^?_1, \ldots, s^?_n$ *and* $s^?$ *such that* $\tau = s^?$ *with* $\mathsf{f} : s^?_1, \ldots, s^?_n \to s^? \in \Gamma$ *and* $\Gamma \vdash \mathsf{t}_i : s^?_i$ *for* $i \in [1, n]$.
6. *If* $\Gamma \vdash \mathsf{v()} : \tau$, *then:*
   (a) *either there are some types* $s^?_1$ *and* s^v *such that* $\tau = s^v$ *with* $\mathsf{v} : (s^?_1)^\star \to s^v \in \Gamma$,
   (b) *or there is a type* $s^?$ *such that* $\tau = s^?$ *with* $\Gamma \vdash \mathsf{v()} : s^v$.

7. *If* $\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \tau$, *then:*

   (a) *there are some types* $s_1^?$ *and* s^v *such that* $\tau = s^v$ *with* $\mathsf{v} : (s_1^?)^\star \to s^v \in \Gamma$ *and* $\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ *and* $\Gamma \vdash \mathsf{t}_n : s^v$, *where* $\mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v$ *or* $t_n = x*$,

   (b) *or there are some types* $s_1^?$ *and* s^v *such that* $\tau = s^v$ *with* $\mathsf{v} : (s_1^?)^\star \to s^v \in \Gamma$ *and* $\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ *and* $\Gamma \vdash \mathsf{t}_n : s_1^?$, *where* $\mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v$ *and* $t_n \neq x*$,

   (c) *or there is a type* $s^?$ *such that* $\tau = s^?$ *with* $\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v$.

8. *If* $\Gamma \vdash \mathsf{x} := \text{`}\mathsf{t} : \tau$ *then* $\tau = wt$ *and there is a type* s^h *such that* $\mathsf{x} : s^h \in \Gamma$ *and* $\Gamma \vdash \mathsf{t} : s^h$.

9. *If* $\Gamma \vdash \text{`}\mathsf{t} : \tau$, *then* $\tau = wt$ *and there is a type* $s^h$ *such that* $\Gamma \vdash \text{`}\mathsf{t} : s^h$.

10. *If* $\Gamma \vdash (\mathsf{t}_1 \ll |s^h|\ \mathsf{t}_2) : \tau$, *then* $\tau = wt$ *with* $\Gamma \vdash \mathsf{t}_1 : s^?$ *and* $\Gamma \vdash \mathsf{t}_2 : s^?$.

11. *If* $\Gamma \vdash (\mathsf{t}_1 \diamond \mathsf{t}_2) : \tau$, *then* $\tau = wt$ *and there is a type* $s^?$ *such that* $\Gamma \vdash \mathsf{t}_1 : s^?$ *and* $\Gamma \vdash \mathsf{t}_2 : s^?$.

12. *If* $\Gamma \vdash (cond_1 \wedge cond_2) : \tau$, *then* $\tau = wt$, $\Gamma \vdash (cond_1) : wt$ *and* $\Gamma \vdash (cond_2) : wt$.

13. *If* $\Gamma \vdash (cond_1 \vee cond_2) : \tau$, *then* $\tau = wt$, $\Gamma \vdash (cond_1) : wt$ *and* $\Gamma \vdash (cond_2) : wt$.

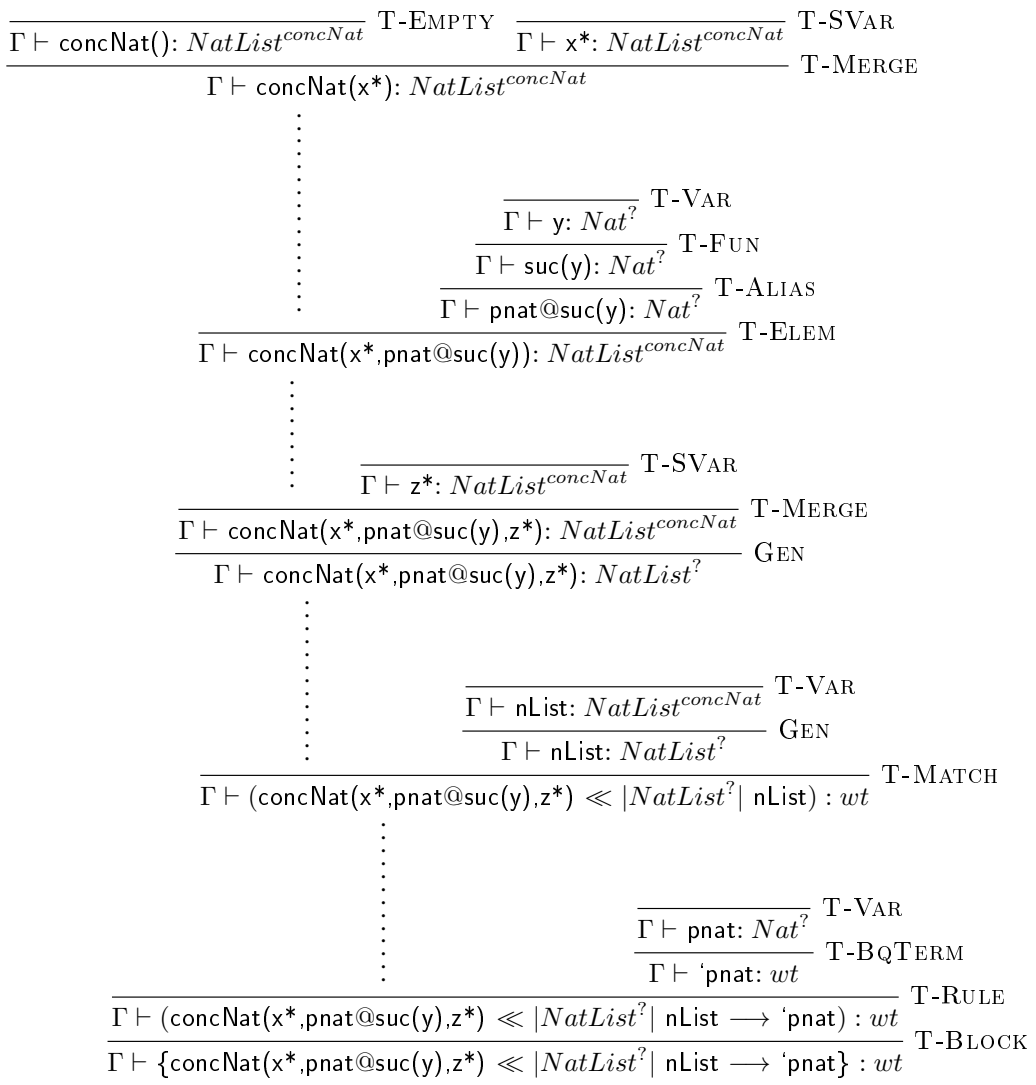
14. *If* $\Gamma \vdash (\mathsf{t}_1;\ldots;\mathsf{t}_n) : \tau$, *then* $\tau = wt$ *and* $\Gamma \vdash \tau_i : wt$ *for* $i \in [1,n]$.

15. *If* $\Gamma \vdash (cond \longrightarrow action) : \tau$, *then* $\tau = wt$ *and* $\Gamma \vdash cond : wt$ *and* $\Gamma \vdash action : wt$.

16. *If* $\Gamma \vdash (rule_1,\ldots,rule_n) : \tau$, *then* $\tau = wt$ *and* $\Gamma \vdash rule_i : wt$ *for* $i \in [1,n]$.

Proof. Immediate from the instance of the conclusion of the appropriate type checking rule combined with the rule [Gen] in Figure 3.7. Furthermore, the rule [Gen] must be considered for clauses 1b, 2b, 6b and 7c. □

The inversion lemma shows how to generate a typing derivation for a valid typing judgment. This derivation constitutes a *proof* of the expression involved in the typing judgment and ensures that the expression is well-typed. We note that well-typed terms may be obtained by more than one typing judgments, since the equality of decorated sorts is considered as in Definition 3.2 instead of the syntactic equality.



$$\Gamma = (\mathsf{suc} : Nat^{?} \to Nat^{?}; \mathsf{concNat} : (Nat^{?})^{\star} \to NatList^{concNat}; \mathsf{x} : NatList^{concNat}; \mathsf{y} : Nat^{?}; \mathsf{z} : NatList^{concNat}; \mathsf{pnat} : Nat^{?}; \mathsf{nList} : NatList^{concNat})$$

Figure 3.8: Example using the type checking algorithm.

Theorem 3.18 (Uniqueness of types)**.** *Each expression e has at most one type in Γ modulo renaming. That is, if e is well-typed, then its type is unique modulo equality of decorated sorts (see Definition 3.2).*

Proof. By induction on the given typing judgment, using the appropriate clause of the inversion lemma (plus the induction hypothesis) for each case. We detail the cases for terms of $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^{\star}, \mathcal{V})$, where the most noteworthy case of this proof is the one applied to non-empty lists.

Case of a variable : $e = \mathsf{x}$

By the inversion lemma 3.17, both clauses 1a and 1b apply:
- By clause 1a, x is of type s^h such that $\mathsf{x} : s^h \in \Gamma$.
- By clause 1b, x can be typed with $s^?$ such that $\Gamma \vdash \mathsf{x}\colon s^v$ for some $\mathsf{v} \in \mathcal{F}^\star$. The definition of equality of decorated sorts tell us that $s^v =_t s^?$, as required.

Case of a function : $e = \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$

By the inversion lemma, only clause 5 applies. Thus, there are some types $s^?$ and $s_i^?$ such that $\tau = s^?$ with $\mathsf{f} : s_1^?,\ldots,s_n^? \to s^? \in \Gamma$ and $\Gamma \vdash \mathsf{t}_i : s_i^?$ for $i \in [1,n]$.

Case of a non-empty list : $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$

By the inversion lemma, clauses 7a, 7b and 7c apply:
- By clause 7a, $\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$ is of type s^v with $\mathsf{v} : (s_1^?)^\star \to s^v \in \Gamma$ for some type $s_1^?$ and $\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1} : s^v$ and t_n is of type s^v.
- By clause 7b, $\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$ is of type s^v with $\mathsf{v} : (s_1^?)^\star \to s^v \in \Gamma$ for some type $s_1^?$ and $\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1} : s^v$ and $\mathsf{t}_n : s_1^?$.
- By clause 7c, $\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$ can be typed with $s^?$ such that $\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v$. The definition of equality of decorated sorts tell us that $s^v =_t s^?$, as required.

□

There remains to show that our type system is safe. This is done in two steps: we first show the soundness of typing with respect to semantics (Theorem 3.19) then we prove that evaluation preserves typing (Theorem 3.21).

Let us focus on the first step and show that typing is correct with respect to semantics, so that a ground well-typed expression is either a value or can take another reduction step according to the evaluation rules. However, expressions such as anti-patterns are treated semantically as in Tom. Informally, having an anti-pattern as an anti-matching condition can only restrict a (possibly infinite) set of values: they describe the acceptance or rejection of conditions. In Tom, variables used in anti-patterns are not bound to values and cannot be used in the right-hand side of the condition, except in some cases where these variables are non-linear and also occur in a pattern that is not an anti-pattern (thus yielding at most one value for this variable).

Theorem 3.19 (Progress)**.** *Let e be a ground well-typed expression (that is, $\Gamma \vdash e : \tau$ for some $\tau \in \mathcal{T}y\backslash\mathcal{X}$ and $\mathcal{V}ar(e) = \emptyset$) such that e is not an anti-pattern. Then either e is a value or there is some value $e' \in \mathcal{V}al$ with either $\emptyset \Vdash e \Downarrow (\rho', e')$ or $\emptyset \Vdash e \Downarrow (\{\rho'_1,\ldots,\rho'_n\}, e')$.*

Proof. By induction on the derivation of $\vdash e : \tau$. The variable, star variable and alias cases cannot occur, since e is ground. The function and (empty) list cases are immediate, since functions and lists are values. For the other cases, we reason as follows:

Case GEN: $e = \mathsf{t} \quad \mathsf{t} : s^v$

The result follows directly from the induction hypothesis.

Case T-MATCH: $e = \mathsf{t}_1 \ll |s^h|\ \mathsf{t}_2 \quad \mathsf{t}_1 : s^? \quad \mathsf{t}_2 : s^?$

According to the Tom code grammar in Definition 3.5, t_2 must be a term and t_1 must be either a term or an anti-pattern. By the induction hypothesis, t_2 is ground, then it is either a function or a list. Then t_2 is a value. If t_1 is a term, then it is either a function or a list. In this case, t_1 is a value and one of the rules [E-MFUNACC], [E-MFUNREJ], [E-MFUNLISTREJ], [E-MLISTACC] and [E-MLISTREJ] applies to e generating either $\emptyset \Vdash e \Downarrow (\{\rho_1, \ldots, \rho_n\}, \mathsf{accept})$ or $\emptyset \Vdash e \Downarrow (\{\emptyset\}, \mathsf{reject})$. If t_1 is an anti-pattern, then, by either [E-MANTIACC] or [E-MANTIREJ], we obtain $\emptyset \Vdash e \Downarrow (\{\rho\}, \mathsf{accept})$ or $\emptyset \Vdash e \Downarrow (\{\emptyset\}, \mathsf{reject})$.

Case T-NUM: $e = \mathsf{t}_1 \diamond \mathsf{t}_2 \quad \mathsf{t}_1 : s^? \quad \mathsf{t}_2 : s^?$

According to the Tom code grammar, t_1 and t_2 must be terms. By the induction hypothesis, t_1 and t_2 are ground, then they are either functions or lists. Then both t_1 and t_2 are values and either rule [E-NUMACC] or rule [E-NUMREJ] applies to e generating either $\emptyset \Vdash e \Downarrow (\{\emptyset\}, \mathsf{accept})$ or $\emptyset \Vdash e \Downarrow (\{\emptyset\}, \mathsf{reject})$.

Case T-CONJ: $e = cond_1 \wedge cond_2 \quad cond_1 : wt \quad cond_2 : wt$

By the induction hypothesis, both $cond_1$ and $cond_2$ are either values or else there are some $\dot{v}_1$, $\dot{v}_2$ such that $\emptyset \Vdash cond_1 \Downarrow (\{\rho_1\}, \dot{v}_1)$ and $\rho_1 \Vdash cond_2 \Downarrow (\{\rho_2\}, \dot{v}_2)$. In all these cases, either rule [E-CONJACC] or rule [E-CONJREJ] applies to e generating respectively $\emptyset \Vdash e \Downarrow (\{\rho_2\}, \mathsf{accept})$ or $\emptyset \Vdash e \Downarrow (\{\emptyset\}, \mathsf{reject})$.

Case T-DISJ: $e = cond_1 \vee cond_2 \quad cond_1 : wt \quad cond_2 : wt$

By the induction hypothesis, both $cond_1$ and $cond_2$ are either values or else there are some $\dot{v}_1$, $\dot{v}_2$ such that $\emptyset \Vdash cond_1 \Downarrow (\{\rho_1, \ldots, \rho_{n_1}\}, \dot{v}_1)$ and $\emptyset \Vdash cond_2 \Downarrow (\{\rho'_1, \ldots, \rho'_{n_2}\}, \dot{v}_2)$. In all these cases, one of the rules [E-EXDISJACC], [E-DISJACC] and [E-DISJREJ] applies to e generating either $\emptyset \Vdash e \Downarrow (\{\rho"_1, \ldots, \rho"_n\}, \mathsf{accept})$ or $\emptyset \Vdash e \Downarrow (\{\emptyset\}, \mathsf{reject})$.

Case T-ACTION: $e = (action_1; \ldots; action_n) \quad action_i : \tau_i \in \mathcal{T}y \backslash \mathcal{X}$, for $i \in [1, n]$

By the induction hypothesis, each $action_i$ is either a value or else there is some $\dot{v}_i$ such that $\emptyset \Vdash action_i \Downarrow (\rho_i, \dot{v}_i)$. In all these cases, by [E-ACTION], $\emptyset \Vdash e \Downarrow (\rho_n, \mathsf{accept})$.

Case T-RULE: $e = cond \longrightarrow action \quad cond : wt \quad action : wt$

By the induction hypothesis, either $cond$ is a value or else there is some $\dot{v}_1$ such that $\emptyset \Vdash cond \Downarrow (\{\rho_1, \ldots, \rho_n\}, \dot{v}_1)$. Still by induction hypothesis, $action$ is either a value or else $\rho_i \Vdash action \Downarrow (\rho'_i, \mathsf{accept})$. In all these cases, rule [E-RULE] applies to e independently of the (number of) evaluation(s) of $action$ and generates either $\emptyset \Vdash e \Downarrow (\emptyset, \mathsf{accept})$ or $\emptyset \Vdash e \Downarrow (\emptyset, \mathsf{reject})$.

Case T-BLOCK: $e = (rule_1; \ldots; rule_n) \quad rule_i : wt$, for $i \in [1, n]$

By the induction hypothesis, each $rule_i$ is either a value or else there is some $\dot{v}_i$ such that $\emptyset \Vdash rule_i \Downarrow (\emptyset, \dot{v}_i)$. In all these cases, either rule [E-BLOCKACC] or rule [E-BLOCKREJ] applies to e generating either $\emptyset \Vdash e \Downarrow (\emptyset, \mathsf{accept})$ or $\emptyset \Vdash e \Downarrow (\emptyset, \mathsf{reject})$. □

Before proving that a step of evaluation preserves types of any Tom expression, we introduce another auxiliary lemma which states that substitution of variables preserve the well-typedness of expressions (the so-called *substitution lemma*).

Lemma 3.20 (Substitution). *If* $\Gamma(\mathsf{x} : s^h) \vdash e : \tau$, *such that* $\tau \in \mathcal{T}y \backslash \mathcal{X}$, *and* $\Gamma \vdash \mathsf{t} : s^h$, *then* $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{t}]e : \tau$.

Proof. Straightforward induction on a derivation of the statement $\Gamma(\mathsf{x} : s^h) \vdash e : \tau$. For a given derivation, we use the appropriate type checking rule (plus the induction hypothesis) for each case. There is one case per typing rule and they are immediate. We show a subset of the possible cases, since the remaining ones are similar.

Case T-VAR: $e = \mathsf{y}$ with $\mathsf{y} : s_1^{h_1} \in \Gamma(\mathsf{x} : s^h)$

There are two subcases to consider depending on whether y is x or another variable. If $\mathsf{y} = \mathsf{x}$, then $s^h = s_1^{h_1}$ and $[\mathsf{x} \mapsto \mathsf{t}]\mathsf{y} = \mathsf{t}$. The required result is then $\Gamma \vdash \mathsf{t} : s^h$ which is among the assumptions of the lemma. Otherwise, $[\mathsf{x} \mapsto \mathsf{t}]\mathsf{y} = \mathsf{y}$, and the desired result is immediate.

Case T-ANTI: $e = !\mathsf{t}_1 \quad \tau = s_1^{h_1} \quad \Gamma(\mathsf{x} : s^h) \vdash \mathsf{t}_1 : s_1^{h_1}$

By the induction hypothesis, $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_1 : s_1^{h_1}$. By the application of [T-ANTI], $\Gamma \vdash ![\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_1 : s_1^{h_1}$. This is precisely the needed result, since, by the definition of substitution, $[\mathsf{x} \mapsto \mathsf{t}]e = ![\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_1$.

Case T-ALIAS: $e = \mathsf{y}@\mathsf{t}_1 \quad \tau = s_1^{h_1} \quad \Gamma(\mathsf{x} : s^h) \vdash \mathsf{t}_1 : s_1^{h_1}$

Since the variables of similar expressions are considered modulo renaming, we assume that $\mathsf{x} \neq \mathsf{y}$ and $\mathsf{y} \in \mathcal{V}ar(\mathsf{t})$. By the induction hypothesis,$\Gamma(\mathsf{x} : s^h) \vdash [\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_1 : s_1^{h_1}$. By the application of [T-ALIAS], $\Gamma(\mathsf{x} : s^h; \mathsf{y} : s_1^{h_1}) \vdash \mathsf{y}@[\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_1 : s_1^{h_1}$, that is $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{t}]e : \tau$.

Case T-MERGE: $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) \quad \tau = s^v$
$\Gamma(\mathsf{x} : s^h) \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v \quad \Gamma(\mathsf{x} : s^h) \vdash \mathsf{t}_n : s^v$

By the induction hypothesis, $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{t}]\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$, i.e. $\Gamma \vdash \mathsf{v}([\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_1,\ldots,[\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_{n-1}) : s^v$, by the definition of substitution. Still by the induction hypothesis, $\Gamma(\mathsf{x} : s^h) \vdash [\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_n : s^v$. By the application of [T-MERGE], $\Gamma \vdash \mathsf{v}([\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_1,\ldots,[\mathsf{x} \mapsto \mathsf{t}]\mathsf{t}_n) : s^v$, that is $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{t}]\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v$ as required. □

As a second step to prove safety, we prove that typing is preserved by evaluation or in other words, if a well-typed expression takes a step of evaluation, then the resulting expression is also well-typed. To this effect, in Figure 3.9 we introduce two more type checking rules for the constants used in evaluation: accept and reject. Despite these constants not belonging to the Tom core abstract syntax (see Definition 3.5), they are used for the evaluation of Tom expressions that are not Tom terms and for this reason they are typed with wt.

$$\frac{}{\Gamma \vdash \mathsf{accept} : wt}\ \text{T-Accept} \qquad \frac{}{\Gamma \vdash \mathsf{reject} : wt}\ \text{T-Reject}$$

Figure 3.9: Type checking rules for constants of evaluation.

Theorem 3.21 (Preservation). *If $\Gamma \vdash e : \tau$ and either $\rho \Vdash e \Downarrow (\rho', e')$ or $\rho \Vdash e \Downarrow (\{\rho'_1, \ldots, \rho'_n\}, e')$ such that e is not an anti-pattern and $e' \in \mathcal{V}al$, then $\Gamma \vdash e' : \tau$, for some $\tau \in \mathcal{T}y \backslash \mathcal{X}$ and ρ, ρ'_i with $i \in [1, n]$.*

Proof. By induction on a derivation of $\Gamma \vdash e : \tau$. The anti-pattern case cannot happen since it does not evaluate as well as the alias case. We proceed by case analysis on the final rule in the derivation and show only a subset of the cases, since the remaining ones are similar.

Case T-Var: $e = \mathsf{x} \quad \tau = s^h$ with $\mathsf{x} : s^h \in \Gamma$

By inspecting the evaluation rules of Figure 3.1, 3.2 and 3.3, we note that there is just one rule, [E-Var], that can be used to derive $\rho \Vdash e \Downarrow (\rho', e')$. The form of this rule tells us that $\rho = \rho'$, $\mathsf{x} \in \mathcal{D}om(\rho)$, $\rho(\mathsf{x}) = e'$ and $e' \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$, i.e., e' is a ground term. By the substitution lemma, $\Gamma \vdash [\mathsf{x} \mapsto e']\mathsf{x} : s^h$, that is $\Gamma \vdash e' : \tau$.

Case Gen: $e = \mathsf{t} \quad \tau = s^? \quad \Gamma \vdash \mathsf{t} : s^v$

One of the rules [E-Var], [E-Val], [E-Fun] and [E-List] can be used to derive $\rho \Vdash e \Downarrow (\rho', e')$. By induction hypothesis, $\Gamma \vdash e' : s^v$. By application of rule [Gen], we obtain $\Gamma \vdash e' : s^?$ as required.

Case T-Fun: $e = \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \quad \tau : s^? \quad \Gamma \vdash \mathsf{t}_i : s_i^?$ for $i \in [1, n]$

The rule [E-Fun] can be used to derive $\rho \Vdash e \Downarrow (\rho', e')$. The form of this rule tells us that $\rho = \rho'$, $\rho \Vdash \mathsf{t}_i \Downarrow (\rho, \dot{u}_i)$ and $e' = \mathsf{f}(\dot{u}_1, \ldots, \dot{u}_n)$. By induction hypothesis, $\Gamma \vdash \dot{u}_i : s_i^?$ for each $\dot{u}_i$. We can apply rule [T-Fun] to conclude that $\Gamma \vdash \mathsf{f}(\dot{u}_1, \ldots, \dot{u}_n) : s^?$, that is $\Gamma \vdash e' : \tau$.

Case T-Match: $e = \mathsf{t}_1 \ll |s^h| \ \mathsf{t}_2 \quad \tau : wt \quad \Gamma \vdash \mathsf{t}_1 : s^? \quad \Gamma \vdash \mathsf{t}_2 : s^?$

One of the rules in either Figure 3.2 or Figure 3.3 can be used to derive $\rho \Vdash e \Downarrow (\{\rho'_1, \ldots, \rho'_n\}, e')$. The form of each one of these rules tells us that either $\dot{v} = \mathsf{accept}$ or $\dot{v} = \mathsf{reject}$, where $e' = \dot{v}$. By application of either rule [T-Accept] or [T-Reject], we obtain $\Gamma \vdash e' : wt$ as required.

Case T-Conj: $e = cond_1 \wedge cond_2 \quad \tau = wt \quad \Gamma \vdash cond_1 : wt \quad \Gamma \vdash cond_2 : wt$

Either rule [E-ConjAcc] or rule [E-ConjRej] can be used to derive $\rho \Vdash cond_1 \wedge cond_2 \Downarrow (\{\rho'\}, \dot{v})$ such that $\dot{v} = \mathsf{accept}$ or $\dot{v} = \mathsf{reject}$, where $e' = \dot{v}$. As in the matching case, by application of either rule [T-Accept] or [T-Reject], we obtain $\Gamma \vdash e' : wt$ as required.

□

3.3.4 Type inference

The type checking algorithm presented in the previous section depends on explicit type annotation even for variables. But, in Tom, only the rank of operators are known. For that reason, we are interested in increasing our type system by defining another algorithm able to infer statically the type of Tom terms. Hence, we develop a more powerful type inference system able of computing a *principal type* for a Tom code in which some of type annotations are left unspecified.

Preliminary definitions

As seen in Section 3.3.3, the set $\mathcal{T}y$ of types used by our type system includes type variables α as uninterpreted types. Actually, type variables are placeholders for ground types whose exact identities are not relevant in typing judgments and can be instantiated with other types. Considering Definition 1.5, we define a *type substitution* σ from type variables to types as done by Benjamin Pierce in his work [Pie02].

Definition 3.22 (Type substitution)**.** *A* type substitution *(or just* substitution*, when it is clear that we are talking about types) is a total function from $\mathcal{X}$ to $\mathcal{T}y$, i.e. from type variables to types.*

We write $\mathcal{D}om(\sigma)$ for the set of type variables appearing on the left-hand sides of pairs in σ, and $\mathcal{R}ange(\sigma)$ for the set of types appearing on the right-hand sides. If $\mathcal{D}om(\sigma) = \{\alpha_1, \ldots, \alpha_n\}$, then we write σ as $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$. We note that the same variable may occur in both the domain and the range of a substitution.

Application of a type substitution to a context is defined by:

$$\sigma(\ldots, \mathsf{x}_1 : \alpha_1, \ldots, \mathsf{x}_n : \alpha_n, \ldots) = (\ldots, \mathsf{x}_1 : \sigma\alpha_1, \ldots, \mathsf{x}_n : \sigma\alpha_n, \ldots)$$

where only variables can be typed with type variables in the context, since the rank of all (algebraic and concrete) operators are type annotations. Type substitutions preserve the validity of typing judgments whose Tom expression involves variables.

Theorem 3.23 (Preservation of typing under type substitution)**.** *Let σ be a type substitution. If $\Gamma \vdash e : \tau$ then $\sigma\Gamma \vdash \sigma e : \sigma\tau$.*

Proof. Straightforward induction on the derivation of the statement $\Gamma \vdash e : \tau$. There is one case per typing rule and they are immediate. For cases [T-Elem] and [T-Merge], their applicability conditions ensure the derivation of well-typed lists since they consider not only the types of arguments but also the structure of these arguments. We just detail these both noteworthy cases of proof as well as the variable case which is the base case.

Case T-Var: $e = \mathsf{x} \quad \tau = \tau_1$ with $\mathsf{x} : \tau_1 \in \Gamma$

There are two subcases to consider depending on whether $\tau_1 \in \mathcal{D}om(\sigma)$ or not. If $\tau_1 \in \mathcal{D}om(\sigma)$, then there is a τ' such that $(\tau_1 \mapsto \tau') \in \sigma$. The required result is then $\sigma\Gamma \vdash \mathsf{x} : \tau'$, since, by the definition of substitution, $\sigma(\Gamma(\mathsf{x} : \tau_1)) = \sigma\Gamma; \sigma(\mathsf{x} : \tau_1) = \sigma\Gamma; \mathsf{x} : \tau'$. Otherwise, $\sigma\tau_1 = \tau_1$ and the desired result is immediate.

Case T-Merge: $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) \qquad \tau = \tau_1$
$\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \tau_1 \quad \Gamma \vdash \mathsf{t}_n : \tau_1 \quad \mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v$ or $\mathsf{t}_n = \mathsf{x}*$

By the induction hypothesis, $\sigma\Gamma \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1})) : \sigma\tau_1$, i.e. $\sigma\Gamma \vdash \mathsf{v}(\sigma\mathsf{t}_1,\ldots,\sigma\mathsf{t}_{n-1}) : \sigma\tau_1$, by the definition of substitution. Still by the induction hypothesis, $\sigma\Gamma \vdash \sigma\mathsf{t}_n : \sigma\tau_1$. Since $\sigma\mathsf{t}_n = \mathsf{t}_n$ and $s^v \notin \mathcal{X}$, then $\mathsf{typeof}(\sigma\Gamma,\sigma\mathsf{t}_n) = s^v$ or $\sigma\mathsf{t}_n = \mathsf{x}*$. By the application of [T-Merge], $\sigma\Gamma \vdash \mathsf{v}(\sigma\mathsf{t}_1,\ldots,\sigma\mathsf{t}_n) : \sigma\tau_1$, that is $\sigma\Gamma \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) : \sigma\tau_1$ as required.

Case T-Elem: $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) \qquad \tau = \tau_1$
$\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \tau_1 \quad \Gamma \vdash \mathsf{t}_n : \tau_2 \quad \mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v$ and $\mathsf{t}_n \neq \mathsf{x}*$

By the induction hypothesis, $\sigma\Gamma \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1})) : \sigma\tau_1$, i.e. $\sigma\Gamma \vdash \mathsf{v}(\sigma\mathsf{t}_1,\ldots,\sigma\mathsf{t}_{n-1}) : \sigma\tau_1$, by the definition of substitution. Still by the induction hypothesis, $\sigma\Gamma \vdash \sigma\mathsf{t}_n : \sigma\tau_2$. Since $\sigma\mathsf{t}_n = \mathsf{t}_n$ and $s^v \notin \mathcal{X}$, then $\mathsf{typeof}(\sigma\Gamma,\sigma\mathsf{t}_n) \neq s^v$ and $\sigma\mathsf{t}_n \neq \mathsf{x}*$. By the application of [T-Elem], $\sigma\Gamma \vdash \mathsf{v}(\sigma\mathsf{t}_1,\ldots,\sigma\mathsf{t}_n) : \sigma\tau_1$, that is $\sigma\Gamma \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) : \sigma\tau_1$ as required. □

The function of decoration cleaning introduced in Definition 3.3 can be applied to type variables by considering type substitutions:

$$|\alpha| = |s^h|, \text{ if there exists a substitution } \sigma \text{ such that } \sigma\alpha = s^h$$

where $\alpha \in \mathcal{X}$.

To determine whether a Tom code containing type variables is well-typed, one must describe how type variables can be instantiated by types, in both expression and context, and build a valid typing derivation.

Definition 3.24 (Solution for typing judgment)**.** *Let Γ be a context and e a Tom expression. A* solution *for (Γ, e) is a pair (σ, τ) such that $\sigma\Gamma \vdash \sigma e : \tau$, where $\tau \in \mathcal{T}y\backslash\mathcal{X}$.*

More than just checking solutions for (Γ, e) by building typing derivations, we are interested in finding these solutions by constraining the type variables which appears in typing derivations. We aim to restrict the set of possible values of the type variables through calculation of a set of *constraints*. Thus, these constraints limit types and consequently base sorts (i.e. algebraic data types) that terms can have.

Definition 3.25 (Constraint set)**.** *Consider the set $\mathcal{T}y$ of types. A* constraint set *$\mathcal{C}$ is a set of constraints defined by the following algebraic grammar:*

$$c ::= \tau_1 =_t \tau_2$$

where $c \in \mathcal{C}$ and $\tau_1, \tau_2 \in \mathcal{T}y$.

A ground constraint *is a constraint with no variables, i.e. a constraint built from ground types.*

A substitution σ is said to satisfy *(or* be a solution of*) an equation $\tau_1 =_t \tau_2$ if $\sigma\tau_1 =_t \sigma\tau_2$. Thus, σ satisfies $\mathcal{C}$ if it satisfies all equations in $\mathcal{C}$. This is written $\sigma \models \mathcal{C}$.*

The set $\mathcal{V}ar(\mathcal{C})$ denotes the set of type variables existing in $\mathcal{C}$.

Type inference rules

A constraint typing judgment for Tom expressions has the form $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ and is defined by a set of inference rules assigning types to Tom expressions, summarized in Figures 3.10 and 3.11. Constraints are calculated according to the application of these rules. Informally, $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ can be read "the expression e is of type τ under assumptions Γ whenever the constraints $\mathcal{C}$ are satisfied". Formally, the typing judgment states that

$$\forall\sigma(\sigma \models \mathcal{C} \Rightarrow \sigma\Gamma \vdash \sigma e : \sigma\tau)$$

In order to infer types of a given expression, all variables must be typed with a type variable before application of inference rules. The rule [CT-Var] states that any type α_1 is a type for x. The rule [CT-SVar] is applied to star variables in the same way. By rule [CT-Anti], the type of an anti-pattern is given by the type of the term being negated. For a term attributed to an alias, both term and alias must be of the same type as stated by rule [CT-Alias]. In the rule [CT-Fun], assuming that $s_1^?, \ldots, s_n^? \to s^?$ is the rank of f in Γ, if each term t_i is of type $s_i^?$ for $i \in [1, n]$ then the application of f to $\mathsf{t}_1,\ldots,\mathsf{t}_n$ must be of type α such that $\alpha =_t s^?$. In the rule [CT-Empty], if v is a variadic function symbol with codomain s^v then a type α which is equal to s^v is a type for an empty list v().

As in the type checking system, the rules applying to list are the most interesting ones. For a non-empty list $\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$ the rule [CT-Elem] is applicable. Therefore, if the element t_n is of type of the domain of v and its head symbol is different from v, then a type α such that $\alpha =_t s^v$ is a type for the original list incremented by t_n. It is still possible to concatenate two lists by application of the rule [CT-Merge]. If these lists are of the same type s^v then a type α such

$$\frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Var} \qquad \frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x*} : \alpha) \vdash_{ct} \mathsf{x*} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-SVar}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash_{ct} !\mathsf{t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Anti} \qquad \frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \quad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x@t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Alias}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_n \bullet \mathcal{C}_n \qquad \mathcal{C} = \{\alpha_0 =_t s^?\} \bigcup_{i=1}^{n} \mathcal{C}_i \cup \{\alpha_i =_t s_i^?\}}{\Gamma(\mathsf{f} : s_1^?, \ldots, s_n^? \to s^?) \vdash_{ct} \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \alpha_0 \bullet \mathcal{C}}\ \text{CT-Fun}$$

$$\frac{\mathcal{C} = \{\alpha_1 =_t s^v\}}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash_{ct} \mathsf{v}() : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Empty}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v, \alpha_2 =_t s_1^?\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \quad \text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v \text{ and } \mathsf{t}_n \neq \mathsf{x*}\ \text{CT-Elem}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_1 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \quad \text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v \text{ or } \mathsf{t}_n = \mathsf{x*}\ \text{CT-Merge}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \quad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} := \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-Ass} \qquad \frac{\Gamma \vdash \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-BqTerm}$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$, $\mathsf{v} \in \mathcal{F}^\star$ and α_i are fresh type variables for $i \in [1, n]$

Figure 3.10: Type inference rules.

that $\alpha =_t s^v$ is a type for the resulting list. The rule [CT-Merge] is also applied to insert a star variable of type s^v into a list with codomain s^v. Tom terms can be introduced by a backquote construct and, in case a term is assigned to a variable, both term and variable must be of the same type as stated by rule [CT-Ass]. The inference of type of a backquoted term is propagated by the rule [CT-BqTerm] collecting the possibly generated constraints.

Our type inference system provides a rule for each "format" of Tom expression even for those which are not Tom terms. The rule [CT-Match] is applied to a pattern matching condition whose operator is declared with type $|\tau|$. It asserts

$$\frac{\Gamma \vdash_{ct} \mathtt{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathtt{t}_2 : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\tau =_t \alpha_1, \alpha_1 =_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathtt{t}_1 \ll |\tau|\ \mathtt{t}_2) : wt \bullet \mathcal{C}}\ \text{CT-Match}$$

$$\frac{\Gamma \vdash_{ct} \mathtt{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathtt{t}_2 : \alpha_1 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathtt{t}_1 \diamond \mathtt{t}_2) : wt \bullet \mathcal{C}}\ \text{CT-Num}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \wedge \ldots \wedge cond_n) : wt \bullet \mathcal{C}}\ \text{CT-Conj}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \vee \ldots \vee cond_n) : wt \bullet \mathcal{C}}\ \text{CT-Disj}$$

$$\frac{\Gamma \vdash_{ct} action_1 : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} action_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (action_1; \ldots; action_n) : wt \bullet \mathcal{C}}\ \text{CT-Action}$$

$$\frac{\Gamma \vdash_{ct} (cond) : wt \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} (action) : wt \bullet \mathcal{C}_2 \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (cond \longrightarrow action) : wt \bullet \mathcal{C}}\ \text{CT-Rule}$$

$$\frac{\Gamma \cup \Gamma_1 \vdash_{ct} rule_1 : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \cup \Gamma_n \vdash_{ct} rule_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \bigcup_{i=1}^{n} \Gamma_i \vdash_{ct} \{rule_1; \ldots; rule_n\} : wt \bullet \mathcal{C}}\ \text{CT-Block}$$

where α_1, α_2 are fresh type variables

Figure 3.11: Type inference rules – continuation.

that if pattern and subject are of a same type α such that $\tau =_t \alpha$ then the condition is well-typed. The rule [CT-Num] is applied similarly to [CT-Match] but since numeric operators are not typed, then only types of terms are restricted by a same type α. Rules [CT-Conj] and [CT-Disj] are applied for a conjunction and a disjunction of conditions, respectively, that are well-typed if all their con-

ditions are well-typed. The remaining rules collect all generated constraints into a final constraint set $\mathcal{C}$. We note that the rule [CT-Block] works with a union of a global context (common to all typing judgments in the premise) and local contexts Γ_i for each $rule_i$. A local context Γ_i contains all variables occurring in *pure* algebraic terms of a Tom program, i.e. in numeric conditions and in patterns of matching conditions of $rule_i$ with their respective annotations. This avoid the need of renaming variables with same name but occurring in different rules $cond \longrightarrow action$. The union $\Gamma \cup \Gamma_i$ comprises all annotated variables and operators occurring in a $rule_i$ expression.

Example 3.26. *Let* $\Gamma = \Gamma' \cup \Gamma_1$ *such that*

$$\Gamma' = (\mathsf{suc} : \ Nat^? \to Nat^?; \mathsf{concNat} : (Nat^?)^\star \to NatList^{concNat}; \mathsf{nList} : \alpha_1)$$

and

$$\Gamma_1 = (\ \mathsf{x} : \alpha_2; \mathsf{y} : \alpha_3; \mathsf{z} : \alpha_4; \mathsf{pnat} : \alpha_5)$$

Considering Example 3.6, the expression

$$\{\mathsf{concNat(x*,pnat@suc(y),z*)} \ll |NatList^?|\ \mathsf{nList} \longrightarrow \mathsf{pnat}\}$$

can be deduced by the tree given in Figure 3.12, where a constraint set $\mathcal{C}$ *is generated.*

Properties of type inference

The type inference rules presented in Figures 3.10 and 3.11 characterize a type inference algorithm able to find a *solution* for a typing judgment based on constraints.

Definition 3.27 (Solution for constraint typing judgment)**.** *Let* Γ *be a context and* e *a Tom expression. Suppose that* $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$*. A* solution *for* $(\Gamma, e, \tau, \mathcal{C})$ *is a pair* (σ, τ') *such that* σ *satisfies* $\mathcal{C}$ *and* $\sigma\tau = \tau'$*, where* $\tau' \in \mathcal{T}y\backslash\mathcal{X}$*.*

The algorithmic approach of the type inference system is equivalent to the declarative one of the type checking system of Section 4.3.2. We demonstrate this equivalence by checking two properties: first, we show that every typing judgment that can be derived from the inference rules also follows from the checking rules (Theorem 3.28), in particular the soundness; then we show that a solution validated by the checking rules can be extended to a solution proposed by the inference rules (Theorem 3.29).

$$\dfrac{\mathcal{C}_{10} = \{\alpha_6 =_t NatList^{concNat}\}}{\Gamma \vdash_{ct} \textsf{concNat()}\colon \alpha_6 \bullet \mathcal{C}_{10}}\ \textsc{CT-Empty}$$

$$\vdots$$

$$\dfrac{\mathcal{C}_{11} = \{\alpha_2 =_t \alpha_6\}}{\Gamma \vdash_{ct} \textsf{x*}\colon \alpha_6 \bullet \mathcal{C}_{11}}\ \textsc{CT-SVar}$$

$$\dfrac{\mathcal{C}_8 = \{\alpha_6 =_t NatList^{concNat}\} \cup \mathcal{C}_{10} \cup \mathcal{C}_{11}}{\Gamma \vdash_{ct} \textsf{concNat(x*)}\colon \alpha_6 \bullet \mathcal{C}_8}\ \textsc{CT-Merge}$$

$$\vdots$$

$$\dfrac{\mathcal{C}_{13} = \{\alpha_3 =_t \alpha_9\}}{\Gamma \vdash_{ct} \textsf{y}\colon \alpha_9 \bullet \mathcal{C}_{13}}\ \textsc{CT-Var}$$

$$\dfrac{\mathcal{C}_{12} = \{\alpha_8 =_t Nat^?, \alpha_9 =_t Nat^?\} \cup \mathcal{C}_{13}}{\Gamma \vdash_{ct} \textsf{suc(y)}\colon \alpha_8 \bullet \mathcal{C}_{12}}\ \textsc{CT-Fun}$$

$$\dfrac{\mathcal{C}_9 = \{\alpha_5 =_t \alpha_8\} \cup \mathcal{C}_{12}}{\Gamma \vdash_{ct} \textsf{pnat@suc(y)}\colon \alpha_8 \bullet \mathcal{C}_9}\ \textsc{CT-Alias}$$

$$\dfrac{\mathcal{C}_6 = \{\alpha_6 =_t NatList^{concNat}, \alpha_8 =_t Nat^?\} \cup \mathcal{C}_8 \cup \mathcal{C}_9}{\Gamma \vdash_{ct} \textsf{concNat(x*,pnat@suc(y))}\colon \alpha_6 \bullet \mathcal{C}_6}\ \textsc{CT-Elem}$$

$$\vdots$$

$$\dfrac{\mathcal{C}_7 = \{\alpha_4 =_t \alpha_6\}}{\Gamma \vdash_{ct} \textsf{z*}\colon \alpha_6 \bullet \mathcal{C}_7}\ \textsc{CT-SVar}$$

$$\dfrac{\mathcal{C}_4 = \{\alpha_6 =_t NatList^{concNat}\} \cup \mathcal{C}_6 \cup \mathcal{C}_7}{\Gamma \vdash_{ct} \textsf{concNat(x*,pnat@suc(y),z*)}\colon \alpha_6 \bullet \mathcal{C}_4}\ \textsc{CT-Merge}$$

$$\vdots$$

$$\dfrac{\mathcal{C}_5 = \{\alpha_1 =_t \alpha_7\}}{\Gamma \vdash_{ct} \textsf{nList}\colon \alpha_7 \bullet \mathcal{C}_5}\ \textsc{CT-Var}$$

$$\dfrac{\mathcal{C}_2 = \{NatList^? =_t \alpha_6, \alpha_6 =_t \alpha_7\} \cup \mathcal{C}_4 \cup \mathcal{C}_5}{\Gamma \vdash_{ct} (\textsf{concNat(x*,pnat@suc(y),z*)} \ll |NatList^?|\ \textsf{nList}) : wt \bullet \mathcal{C}_2}\ \textsc{CT-Match}$$

$$\vdots$$

$$\dfrac{\mathcal{C}_{14} = \{\alpha_5 =_t \alpha_{10}\}}{\Gamma \vdash_{ct} \textsf{pnat}\colon \alpha_{10} \bullet \mathcal{C}_{14}}\ \textsc{CT-Var}$$

$$\dfrac{\mathcal{C}_3 = \mathcal{C}_{14}}{\Gamma \vdash_{ct} \textsf{`pnat}\colon wt \bullet \mathcal{C}_3}\ \textsc{CT-BqTerm}$$

$$\dfrac{\mathcal{C}_1 = \mathcal{C}_2 \cup \mathcal{C}_3}{\Gamma \vdash_{ct} (\textsf{concNat(x*,pnat@suc(y),z*)} \ll |NatList^?|\ \textsf{nList} \longrightarrow \textsf{`pnat}) : wt \bullet \mathcal{C}_1}\ \textsc{CT-Rule}$$

$$\dfrac{\mathcal{C} = \mathcal{C}_1}{\Gamma \vdash_{ct} \{\textsf{concNat(x*,pnat@suc(y),z*)} \ll |NatList^?|\ \textsf{nList} \longrightarrow \textsf{`pnat}\} : wt \bullet \mathcal{C}}\ \textsc{CT-Block}$$

Figure 3.12: Example using the type inference rules.

Theorem 3.28 (Soundness of constraint typing)**.** *Suppose that* $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ *is a valid sequent. If* (σ, τ') *is a solution for* $(\Gamma, e, \tau, \mathcal{C})$*, then it is also a solution for* (Γ, e) *(i.e. e is well-typed in* Γ*).*

Proof. By induction on the given constraint typing derivation for $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$. We just detail the most noteworthy cases of this proof.

Case CT-Elem: $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$, $\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1$, $\Gamma \vdash_{ct} \mathsf{t}_n : \alpha_2 \bullet \mathcal{C}_2$, $\mathcal{C} = \{\alpha_1 =_t s^v, \alpha_2 =_t s_1^?\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$

We are given that (σ, s^v) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e, \alpha_1, \mathcal{C})$, that is, σ satisfies $\mathcal{C}$ and $\sigma\alpha_1 = s^v$. Thus, $\sigma\alpha_1 =_t s^v$ and $\sigma\alpha_2 =_t s^v$. Since σ satisfies $\mathcal{C}_1$ and $\mathcal{C}_2$, $(\sigma, \sigma\alpha_1)$ and $(\sigma, \sigma\alpha_2)$ are solutions for $(\Gamma, \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}), \alpha_1, \mathcal{C}_1)$ and $(\Gamma, \mathsf{t}_n, \alpha_2, \mathcal{C}_2)$, respectively. By induction hypothesis, we have $\sigma\Gamma \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1})) : \sigma\alpha_1$ and $\sigma\Gamma \vdash \sigma\mathsf{t}_n : s^v$. Since $\sigma\alpha_2 =_t s_1^?$, by [Gen] we obtain $\sigma\Gamma \vdash \mathsf{t}_n : s_1^?$. Then, by [T-Elem] we obtain $\sigma(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) : s^v$, that is, (σ, s^v) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e)$ as required.

Case CT-Merge: $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$, $\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1$, $\Gamma \vdash_{ct} \mathsf{t}_n : \alpha_1 \bullet \mathcal{C}_2$, $\mathcal{C} = \{\alpha_1 =_t s^v\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$

We are given that (σ, s^v) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e, \alpha_1, \mathcal{C})$, that is, σ satisfies $\mathcal{C}$ and $\sigma\alpha_1 = s^v$. Since σ satisfies $\mathcal{C}_1$ and $\mathcal{C}_2$, (σ, s^v) is a solution for both $(\Gamma, \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}), \alpha_1, \mathcal{C}_1)$ and $(\Gamma, \mathsf{t}_n, \alpha_1, \mathcal{C}_2)$. By induction hypothesis, we have $\sigma\Gamma \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1})) : s^v$ and $\sigma\Gamma \vdash \sigma\mathsf{t}_n : s^v$. By [T-Merge] we obtain $\sigma(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) : s^v$, that is, (σ, s^v) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e)$ as required.

Case CT-Match: $(\mathsf{t}_1 \ll |\tau| \; \mathsf{t}_2)$, $\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1$, $\Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2$, $\mathcal{C} = \{\tau =_t \alpha_1, \tau\alpha_1 =_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$

We are given that (σ, wt) is a solution for $(\Gamma, e, wt, \mathcal{C})$, that is, σ satisfies $\mathcal{C}$ and $\sigma wt = wt$. Thus, $\sigma\tau =_t \sigma\alpha_1$ and $\sigma\alpha_1 =_t \sigma\alpha_2$. Since σ satisfies $\mathcal{C}_1$ and $\mathcal{C}_2$, $(\sigma, \sigma\alpha_1)$ and $(\sigma, \sigma\alpha_2)$ are solutions for $(\Gamma, \mathsf{t}_1, \alpha_1, \mathcal{C}_1)$ and $(\Gamma, \mathsf{t}_2, \alpha_2, \mathcal{C}_2)$, respectively. By induction hypothesis, we have $\sigma\Gamma \vdash \sigma\mathsf{t}_1 : \sigma\alpha_1$ and $\sigma\Gamma \vdash \sigma\mathsf{t}_2 : \sigma\alpha_2$. Let $\sigma\tau = s^h$ such that $h \in \mathcal{F}^\star \cup \{?\}$. In case $\sigma\alpha_1 = s^{v_1}$ and/or $\sigma\alpha_2 = s^{v_2}$, by [Gen] we obtain $\sigma\Gamma \vdash \mathsf{t}_1 : s^?$ and $\Gamma \vdash \mathsf{t}_2 : s^?$. By [T-Match] we obtain $\sigma\Gamma \vdash \sigma(\mathsf{t}_1 \ll |\tau| \; \mathsf{t}_2) : wt$, that is, (σ, wt) is a solution for (Γ, e) as required.

□

Theorem 3.29 (Completeness of constraint typing)**.** *Suppose that* $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ *is obtained from the last subderivation* π*. We denote* $\mathcal{V}ar(\pi)$ *the set of all type variables appearing in* π *and we also denote* $\sigma\backslash\mathcal{V}ar(\pi)$ *the substitution that behaves like* σ *except for those variables belonging to* $\mathcal{V}ar(\pi)$ *for which the substitution is undefined. If* (σ, τ') *is a solution for* (Γ, e) *and* $\mathcal{D}om(\sigma) \cap \mathcal{V}ar(\pi) = \emptyset$*, then there is some solution* (σ', τ') *for* $(\Gamma, e, \tau, \mathcal{C})$ *such that* $\sigma'\backslash\mathcal{V}ar(\pi) = \sigma$*.*

Proof. By induction on the given constraint typing derivation in normal form, but we must take care with fresh names of variables. We just detail the most noteworthy cases of this proof.

Case CT-Elem: $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$
$\pi_1 = \Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1$ $\quad \pi_2 = \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_2 \bullet \mathcal{C}_2$
$\mathcal{C} = \{\alpha_1 =_t s^v, \alpha_2 =_t s_1^?\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$ $\quad \mathcal{V}ar(\pi) = \{\alpha_1, \alpha_2\}$
$\mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v$

We are given that (σ, s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e)$. By the inversion lemma 3.17, clauses 7a, 7b and 7c apply. We reason by subcases as follows:

1. Subcase [T-Merge]: if $h = v$, then there is a type $s_1^?$ such that $\mathsf{v} : (s_1^?)^\star \to s^v \in \sigma\Gamma$ with $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ and $\sigma\Gamma \vdash \mathsf{t}_n : s^v$. But, since $\mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v$, $\sigma\Gamma \vdash \mathsf{t}_n : s^v$ cannot be derived even from [Gen]. Thus this is not a relevant subcase.
2. Subcase [T-Elem]: if $h = v$, then there is a type $s_1^?$ such that $\mathsf{v} : (s_1^?)^\star \to s^v \in \sigma\Gamma$ with $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ and $\sigma\Gamma \vdash \mathsf{t}_n : s_1^?$. By induction hypothesis, there are solutions (σ_1, s^v) for $(\Gamma, \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}), \alpha_1, \mathcal{C}_1)$ and $(\sigma_2, s_1^?)$ for $(\Gamma, \mathsf{t}_n, \alpha_2, \mathcal{C}_2)$ such that $\mathcal{D}om(\sigma_1)\backslash\mathcal{V}ar(\pi_1) = \sigma = \mathcal{D}om(\sigma_2)\backslash\mathcal{V}ar(\pi_2)$. Define $\sigma' = \sigma_1 \cup \sigma_2 \cup \sigma$. Thus σ' satisfies $\mathcal{C}$ and $\sigma'\alpha_1 = s^v$, that is, (σ', s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e, \alpha_1, \mathcal{C})$ as required.
3. Subcase [Gen]: if $h = ?$, then $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v$. The rest of the proof is similar to previous subcase 2.

Case CT-Merge: $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$
$\pi = \Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1$ $\quad \pi = \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_1 \bullet \mathcal{C}_2$
$\mathcal{C} = \{\alpha_1 =_t s^v\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$ $\quad \mathcal{V}ar(\pi) = \{\alpha_1\}$
$\mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v$

We are given that (σ, s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e)$. By the inversion lemma 3.17, clauses 7a, 7b and 7c apply, as in previous case. We reason by subcases as follows:

1. Subcase [T-Merge]: if $h = v$, then there is a type $s_1^?$ such that $\mathsf{v} : (s_1^?)^\star \to s^v \in \sigma\Gamma$ with $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ and $\sigma\Gamma \vdash \mathsf{t}_n : s^v$. By induction hypothesis, there are solutions (σ_1, s^v) for $(\Gamma, \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}), \alpha_1, \mathcal{C}_1)$ and (σ_2, s^v) for $(\Gamma, \mathsf{t}_n, \alpha_1, \mathcal{C}_2)$ such that $\mathcal{D}om(\sigma_1)\backslash\mathcal{V}ar(\pi_1) = \sigma = \mathcal{D}om(\sigma_2)\backslash\mathcal{V}ar(\pi_2)$. So, $\sigma_1\alpha_1 = s^v = \sigma_2\alpha_1$. Define $\sigma' = \sigma_1 \cup \sigma_2 \cup \sigma$. Thus σ' satisfies $\mathcal{C}$ and $\sigma'\alpha_1 = s^v$, that is, (σ', s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e, \alpha_1, \mathcal{C})$ as required.
2. Subcase [T-Elem]: if $h = v$, then there is a type $s_1^?$ such that $\mathsf{v} : (s_1^?)^\star \to s^v \in \sigma\Gamma$ with $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ and $\sigma\Gamma \vdash \mathsf{t}_n : s_1^?$, where $\mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v$. But, since $\mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v$, this is not a relevant subcase.
3. Subcase [Gen]: if $h = ?$, then $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v$. The rest of the proof is similar to previous subcase 1.

Case CT-Match: $(\mathsf{t}_1 \ll |\tau| \ \mathsf{t}_2)$
$\pi_1 = \Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1$ $\quad \pi_2 = \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2$
$\mathcal{C} = \{\tau =_t \alpha_1, \alpha_1 =_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$ $\quad \mathcal{V}ar(\pi) = \{\alpha_1, \alpha_2\}$

We are given that (σ, wt) is a solution for (Γ, e). By the inversion lemma, we obtain $\sigma\Gamma \vdash \mathtt{t}_1 : s^?$ and $\sigma\Gamma \vdash \mathtt{t}_2 : s^?$ such that $\sigma\tau =_t s^?$. By induction hypothesis, there are solutions $(\sigma_1, s^?)$ for $(\Gamma, \mathtt{t}_1, \alpha_1, \mathcal{C}_1)$ and $(\sigma_2, s^?)$ for $(\Gamma, \mathtt{t}_2, \alpha_2, \mathcal{C}_1)$ such that $\mathcal{D}om(\sigma_1)\backslash\mathcal{V}ar(\pi_1) = \sigma = \mathcal{D}om(\sigma_2)\backslash\mathcal{V}ar(\pi_2)$. So, $\sigma_1\alpha_1 = s^? = \sigma_2\alpha$. Define $\sigma' = \sigma_1 \cup \sigma_2 \cup \sigma$. Thus σ' satisfies $\mathcal{C}$ and $\sigma' wt = wt$, that is, (σ', wt) is a solution for $(\Gamma, e, wt, \mathcal{C})$ as required. □

3.3.5 Constraint resolution

We have done the generation of equality constraints by application of type inference rules. In this section we shall focus on how to solve equality constraints. First and foremost, we consider the initial equality constraint set in *canonical form.*

Definition 3.30 (Canonical form of equality constraint set)**.** *An equality constraint set $\mathcal{C}$ is said to be in* canonical form *if it satisfies the following property:*

$$\forall \alpha \in \mathcal{X}, \forall s^? \in \mathcal{T}y\backslash(\mathcal{X} \cup \{wt\}), \forall v \in \mathcal{F}^\star, (\alpha =_t s^v \in \mathcal{C} \Rightarrow \alpha =_t s^? \notin \mathcal{C})$$

In Figure 3.13 we present a constraint resolution algorithm solveEqConstraints composed of a set of conditional rewrite rules. It produces a set *err* of type errors possibly found and a substitution σ that satisfies an initial input constraint set. The algorithm is based on the idea, due to the independent works of Hindley [Hin69] and Milner [Mil78], of using unification (see Definition 1.8) to check that a constraint set has a solution and, if so, to find a *principal solution* to it.

Definition 3.31 (Principal solution for constraint typing judgment)**.** *Let Γ be a context and e a Tom expression. Suppose that $\Gamma \vdash_{ct} \mathsf{e} : \tau \bullet \mathcal{C}$. A* principal solution *for $(\Gamma, e, \tau, \mathcal{C})$ is a solution (σ, τ') such that, whenever (σ_1, τ_1) is also a solution for $(\Gamma, e, \tau, \mathcal{C})$, we have $\sigma \lesssim \sigma_1$ (see Definition 1.7).*

A principal solution (σ, τ) for $(\Gamma, e, \tau, \mathcal{C})$ is called a principal type *of e under Γ.*

The rules of Figure 3.13 are applied to all elements of a constraint set $\mathcal{C}$ to perform unification. It considers the commutative property of the $=_t$ operator. For an equation of the form $\tau_1 =_t \tau_2$, each possible instance of τ_1 and τ_2 is considered. An auxiliary algorithm isEq, made of the rewrite rules presented in Figure 3.14, is called each time a ground equation is found in order to verify if the equality of decorated sorts (as described in Definition 3.2) holds for τ_1 and τ_2. Otherwise, when the constraint is not a ground equation, σ is extended, except when a type error is found.

Initially, $\mathcal{C}$ has only equations with two types τ_1 and τ_2 and the solution σ is empty. In order to keep a track of type errors found during constraint resolution, the rules increment a set *err*, initially empty, of pairs of types for which the equality of decorated sorts does not hold. Each element of *err* produces an error message and states that $\mathcal{C}$ has no solution, i.e. that the solution σ generated is not valid. This approach of collection of pairs of types corresponds

$$
\begin{array}{lllll}
(1) & \tau =_t \tau, err, \sigma & \Longrightarrow & err, \sigma & \\
(2) & s_1^{h_1} =_t s_2^{h_2}, err, \sigma & \Longrightarrow & \mathsf{isEq}(s_1^{h_1}, s_2^{h_2}) \cup err,\ \sigma & \\
(3a) & s_1^{h_1} =_t \alpha_1, err, \sigma & \Longrightarrow & err, \downarrow([\alpha_1 \mapsto s_1^{h_1}] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_1 \\
(3b) & s_1^{h_1} =_t \alpha_1, err, \sigma & \Longrightarrow & \mathsf{isEq}(s_1^{h_1}, s_2^{h_2}) \cup err, \sigma & \text{if } \sigma\alpha_1 = s_2^{h_2} \\
(3c) & s_1^{h_1} =_t \alpha_1, err, \sigma & \Longrightarrow & err, \downarrow([\alpha_2 \mapsto s_1^{h_1}] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_2 \\
(4a) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow([\alpha_1 \mapsto \alpha_2] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_1 \wedge \sigma\alpha_2 = \alpha_2 \\
(4b) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow([\alpha_1 \mapsto s_1^{h_1}] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_1 \wedge \sigma\alpha_2 = s_1^{h_1} \\
(4c) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow([\alpha_1 \mapsto \alpha_3] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_1 \wedge \sigma\alpha_2 = \alpha_3 \\
(4d) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & \mathsf{isEq}(s_1^{h_1}, s_2^{h_2}) \cup err, \sigma & \text{if } \sigma\alpha_1 = s_1^{h_1} \wedge \sigma\alpha_2 = s_2^{h_2} \\
(4e) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow([\alpha_3 \mapsto s_1^{h_1}] \circ \sigma) & \text{if } \sigma\alpha_1 = s_1^{h_1} \wedge \sigma\alpha_2 = \alpha_3 \\
(4f) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow([\alpha_3 \mapsto \alpha_4] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_3 \wedge \sigma\alpha_2 = \alpha_4
\end{array}
$$

where $h_1, h_2 \in \mathcal{F}^\star \cup \{?\}$

Figure 3.13: The rules of algorithm solveEqConstraints for solving a constraint set $\mathcal{C}$.

$$
\begin{array}{lllll}
(1a) & s_1^?, s_2^h & \Longrightarrow & \{\} & \text{if } s_1 = s_2 \\
(1b) & s_1^?, s_2^h & \Longrightarrow & \{(s_1^?, s_2^h)\} & \text{if } s_1 \neq s_2 \\
(2a) & s_1^h, s_2^? & \Longrightarrow & \{\} & \text{if } s_1 = s_2 \\
(2b) & s_1^h, s_2^? & \Longrightarrow & \{(s_1^h, s_2^?)\} & \text{if } s_1 \neq s_2 \\
(3a) & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{\} & \text{if } s_1 = s_2 \wedge v_1 = v_2 \\
(3b) & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{(s_1^{v_1}, s_2^{v_2})\} & \text{if } s_1 \neq s_2 \vee v_1 \neq v_2
\end{array}
$$

where $h \in \mathcal{F}^\star \cup \{?\}$ and $v_1, v_2 \in \mathcal{F}^\star$

Figure 3.14: The rules of algorithm isEq for fail detection in a equality constraint set $\mathcal{C}$.

to a Java-like approach since it allows the type system to raise all type errors found during constraint resolution (instead of only the first one). Nevertheless, another possible approach would be to stop the constraint resolution when err becomes non-empty for the first time, i.e. to add a condition as $\mathsf{isEq}(\tau_1, \tau_2) \neq \emptyset$ for each rule.

The operational behavior of the constraint resolution consists in folding the algorithm of Figure 3.13 over the initial constraint set $\mathcal{C}$ with an empty set err of type errors and an empty solution σ:

foldl solveEqConstraints $(err, \sigma)\ \mathcal{C}$

The application of σ to the current equation is done indirectly by conserving σ in a *saturated form*.

Definition 3.32 (Saturated form)**.** *Let Γ be a context, e a Tom expression and σ a solution for $(\Gamma, e, \tau, \mathcal{C})$. σ is said to be in* saturated form*, denoted by $\downarrow\sigma$, if it satisfies the following property:*

$$\forall \alpha \in \mathcal{X}, (\alpha \in \mathcal{D}om(\sigma) \Rightarrow \alpha \notin \mathcal{R}ange(\sigma))$$

The notion of unifier such as in Definition 1.8 is applied to constraint sets considering equality of decorated sorts (see Definition 3.2). Based on unification, we define a *solution* and a *principal solution* for a constraint set $\mathcal{C}$.

Definition 3.33 (Solution for constraint set)**.** *Let $\mathcal{C}$ be a constraint set composed of equality constraints $\tau_1 =_t \tau_2$. A* solution *of an equality constraint $\tau_1 =_t \tau_2$ is a substitution σ such that $\sigma\tau_1 =_t \sigma\tau_1$. A substitution is solution of $\mathcal{C}$ if it is solution of every equality constraint in $\mathcal{C}$.*

Definition 3.34 (Principal solution for constraint set)**.** *A* principal solution *for a constraint set $\mathcal{C}$ is a solution σ of $\mathcal{C}$ such that $\sigma \lesssim \delta$ for every solution δ of $\mathcal{C}$ (see Definition 1.7).*

Here comes the definition of the constraint resolution algorithm solveEqConstraints.

Definition 3.35 (Equality constraint resolution algorithm)**.** *The algorithm* solveEqConstraints *takes as input a 3-tuple $(\mathcal{C}, err, \sigma)$, where $\mathcal{C}$ is an equality constraint set, err is a set of type errors initially empty and σ is a substitution initially empty.*

At every step, the algorithm takes the first equation of $\mathcal{C}$ and tries to apply one of the conditional rewrite rules of Figure 3.13 and then takes the second equation to do the same and so forth until it reaches the end of $\mathcal{C}$. For an equation between ground types and/or type variables mapped by σ to ground types, the first applicable rule of the algorithm isEq *for fail detection is applied to the equality constraint. It returns either an empty set or a singleton set with a pair of types. If the resulting set is non-empty, then the algorithm raises out the respective type error found and computes the union of this set with err.*

At the end of $\mathcal{C}$, if err is empty, then the generated σ constitutes a solution for the original $\mathcal{C}$. Otherwise, all error messages collected during the resolution have been raised and the algorithm fails because either the original $\mathcal{C}$ has no solution or more type annotations must be provided in order to find a principal solution for the original $\mathcal{C}$.

The algorithm solveEqConstraints is non-recursive and corresponds to a derecursivated version of the classical syntactic unification algorithm, except for the manipulation of a set err of type errors. The algorithm stops when the end of

the constraint set is reached. If $err \neq \emptyset$, the algorithm is considered to have failed. It entails that the computed substitution σ is rejected and that all errors messages collected are raised. Otherwise, σ is applied to the Tom expression considered in the constraint typing judgment.

Example 3.36. *Considering Example 3.26, in the constraint typing judgment*

$$\Gamma \vdash_{ct} \{\mathsf{concNat(x*,pnat@suc(y),z*)} \ll |NatList^{?}|\ \mathsf{nList} \longrightarrow \mathsf{pnat}\} : wt \bullet \mathcal{C}$$

the generated constraint set in canonical form is $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$ *where*

$\mathcal{C}_1 = \{\alpha_6 =_t \alpha_7, \alpha_6 =_t NatList^{concNat}, \alpha_8 =_t Nat^{?}\}$
$\mathcal{C}_2 = \{\alpha_2 =_t \alpha_6, \alpha_5 =_t \alpha_8, \alpha_9 =_t Nat^{?}, \alpha_3 =_t \alpha_9\}$
$\mathcal{C}_3 = \{\alpha_4 =_t \alpha_6, \alpha_1 =_t \alpha_7, \alpha_5 =_t \alpha_{10}\}$

Let $err_1 = \{\}$ *and* $\sigma_1 = \{\}$. *Applying* solveEqConstraints *to* $(\mathcal{C}_1, err_1, \sigma_1)$, *we obtain the following reduction sequence* seq_1*:*

$(\{\alpha_6 =_t \alpha_7, \alpha_6 =_t NatList^{concNat}, \alpha_8 =_t Nat^{?}\}, err_1, \sigma_1)$
$\longrightarrow_{(4a)} (\{\alpha_6 =_t NatList^{concNat}, \alpha_8 =_t Nat^{?}\}, err_1, \{\alpha_6 \mapsto \alpha_7\} \circ \sigma_1)$
$\longrightarrow_{(3c)} (\{\alpha_8 =_t Nat^{?}\}, err_1, \{\alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto NatList^{concNat}\} \circ \sigma_1)$
$\longrightarrow_{(3a)} (\{\}, err_1, \{\alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto NatList^{concNat}, \alpha_8 \mapsto Nat^{?}\} \circ \sigma_1)$

Let $err_2 = \{\}$ *and* $\sigma_2 = \{\alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto NatList^{concNat}, \alpha_8 \mapsto Nat^{?}\}$. *Applying* solveEqConstraints *to* $(\mathcal{C}_2, err_2, \sigma_2)$, *we obtain the following reduction sequence* seq_2*:*

$(\{\alpha_2 =_t \alpha_6, \alpha_5 =_t \alpha_8, \alpha_9 =_t Nat^{?}, \alpha_3 =_t \alpha_9\}, err_1, \sigma_1)$
$\longrightarrow_{(4b)} (\{\alpha_5 =_t \alpha_8, \alpha_9 =_t Nat^{?}, \alpha_3 =_t \alpha_9\}, err_2, \{\alpha_2 \mapsto NatList^{concNat}\} \circ \sigma_2)$
$\longrightarrow_{(4b)} (\{\alpha_9 =_t Nat^{?}, \alpha_3 =_t \alpha_9\}, err_2, \{\alpha_2 \mapsto NatList^{concNat}, \alpha_5 \mapsto Nat^{?}\} \circ \sigma_2)$
$\longrightarrow_{(3a)} (\{\alpha_3 =_t \alpha_9\}, err_2, \{\alpha_2 \mapsto NatList^{concNat}, \alpha_5 \mapsto Nat^{?}, \alpha_9 \mapsto Nat^{?}\} \circ \sigma_2)$
$\longrightarrow_{(4b)} (\{\}, err_2, \{\alpha_2 \mapsto NatList^{concNat}, \alpha_5 \mapsto Nat^{?}, \alpha_9 \mapsto Nat^{?}, \alpha_3 \mapsto Nat^{?}\} \circ \sigma_2)$

Let $err_3 = \{\}$ *and* $\sigma_3 = \{\alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto NatList^{concNat}, \alpha_8 \mapsto Nat^{?}, \alpha_2 \mapsto NatList^{concNat}, \alpha_5 \mapsto Nat^{?}, \alpha_9 \mapsto Nat^{?}, \alpha_3 \mapsto Nat^{?}\}$. *Applying* solveEqConstraints *to* $(\mathcal{C}_3, err_3, \sigma_3)$, *we obtain the following reduction sequence* seq_3*:*

$(\{\alpha_4 =_t \alpha_6, \alpha_1 =_t \alpha_7, \alpha_5 =_t \alpha_{10}\}, err_3, \sigma_3)$
$\longrightarrow_{(4b)} (\{\alpha_1 =_t \alpha_7, \alpha_5 =_t \alpha_{10}\}, err_3, \{\alpha_4 \mapsto NatList^{concNat}\} \circ \sigma_3)$
$\longrightarrow_{(4b)} (\{\alpha_5 =_t \alpha_{10}\}, err_3, \{\alpha_4 \mapsto NatList^{concNat}, \alpha_1 \mapsto NatList^{concNat}\} \circ \sigma_3)$
$\longrightarrow_{(4b)} (\{\}, err_3, \{\alpha_4 \mapsto NatList^{concNat}, \alpha_1 \mapsto NatList^{concNat}, \alpha_{10} \mapsto Nat^{?}\} \circ \sigma_3)$

After the reduction sequence $seq_1; seq_2; seq_3$, *the constraint set* $\mathcal{C}$ *is empty and then the algorithm stops. Moreover,* $\{\alpha_1 \mapsto NatList^{concNat}, \alpha_2 \mapsto NatList^{concNat}, \alpha_3 \mapsto Nat^{?}, \alpha_4 \mapsto NatList^{concNat}, \alpha_5 \mapsto Nat^{?}, \alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto NatList^{concNat}, \alpha_8 \mapsto Nat^{?}, \alpha_9 \mapsto Nat^{?}, \alpha_{10} \mapsto Nat^{?}\}$ *is generated as a solution for* $(\Gamma, \{\mathsf{concNat(x*,pnat@suc(y),z*)} \ll |NatList^{?}|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}, wt, \mathcal{C})$, *since* $err_1 \cup err_2 \cup err_3 = \emptyset$.

The algorithm solveEqConstraints always terminates, failing when given a non-unifiable constraint set as input and otherwise returning a principal solution for the initial constraint set. Because of the use of **foldl**, the termination of constraint resolution is equivalent to the termination of solveEqConstraints.

Theorem 3.37 (Termination of constraint resolution)**.**

1. solveEqConstraints *halts, either failing or by returning a solution for the initial constraint set* $\mathcal{C}$*,*
2. *if* solveEqConstraints *returns both a substitution* σ *and an empty set* err*, then* σ *is a solution for* $\mathcal{C}$*,*
3. *if* solveEqConstraints *returns a solution* σ *for* $\mathcal{C}$ *and if* δ *is also a solution for* $\mathcal{C}$*, then* σ *is the principal solution for* $\mathcal{C}$*, i.e.* $\sigma \lesssim \delta$*.*

Proof.

For part [1], each rule of Figure 3.13 terminates immediately and the number of type constraints to be considered by **foldl** decreases trivially until it reaches the end of $\mathcal{C}$.

For part [2], we aim to show that in an error-free run of the algorithm (i.e. $err = \emptyset$ is returned) for a given constraint set $\mathcal{C} = \{c\} \cup \mathcal{C}'$ such that σ satisfies $\mathcal{C}'$, then the algorithm produces a substitution $\delta =\downarrow (\phi \cup \sigma)$ and δ satisfies $\mathcal{C}$. This is done by induction on the number of times **foldl** calls solveEqConstraints which is equivalent to the number of elements of $\mathcal{C}$. We just detail the base case and the most noteworthy cases of this proof since the others are similar.

Case length $\mathcal{C} = 0$: $\quad \mathcal{C} = \{\}$

Since $\mathcal{C}$ is empty, then $err = \emptyset$ and $\sigma = [\,]$. Define $\delta = [\,] \circ \sigma$ which satisfies $\mathcal{C}$ as required.

Case length $\mathcal{C} = n$:

Subcase (2) : $\quad \mathcal{C} = \{s_1^{h_1} =_t s_2^{h_2}\} \cup \mathcal{C}'$

By induction hypothesis, σ satisfies $\mathcal{C}'$ and $err = \emptyset$. If $\mathsf{isEq}(s_1^{h_1}, s_2^{h_2})$ returns $\{\}$, this means that $s_1^{h_1} =_t s_2^{h_2}$. Define $\delta = [\,] \circ \sigma$ which satisfies $\mathcal{C}$ as required.

Subcase (5i) : $\quad \mathcal{C} = \{\alpha_1 =_t \alpha_2\} \cup \mathcal{C}'$
$\sigma\alpha_1 = \alpha_3 \qquad\qquad \sigma\alpha_2 = \alpha_4$

By induction hypothesis, σ satisfies $\mathcal{C}'$ and $err = \emptyset$. Since $\delta =\downarrow ([\alpha_3 \mapsto \alpha_4] \circ \sigma)$, then $\delta\alpha_1 = \alpha_4 = \delta\alpha_2$. Thus $\delta\alpha_1 =_t \delta\alpha_2$, that is, δ satisfies $\mathcal{C}$ as required.

For part [3], we aim to show that if δ satisfies $\mathcal{C}$, then there is a substitution ϕ such that $\delta =_t \phi \circ \sigma$. We proceed again by induction on the number of elements of $\mathcal{C}$. We just detail the base case and a noteworthy case of this proof since the others are similar.

Case length $\mathcal{C} = 0$: $\quad \mathcal{C} = \{\}$

Since $\mathcal{C}$ is empty, then $err = \emptyset$ and $\sigma = [\,]$. Define $\delta = [\,] \circ \sigma$. Thus, $\sigma \lesssim \delta$ as required.

Case length $\mathcal{C} = n$:

Subcase (3a) : $\mathcal{C} = \{s_1^{h_1} =_t \alpha_1\} \cup \mathcal{C}' \quad \sigma\alpha_1 = \alpha_1$

Since δ satisfies $\{s_1^{h_1} =_t \alpha_1\}$, we have $\delta s_1^{h_1} =_t \delta\alpha_1$. By inducton hypothesis, σ' satisfies $\mathcal{C}'$ and $\delta =_t \phi \circ \sigma'$ for some ϕ. Since solveEqConstraints returns $\sigma =\downarrow ([\alpha_1 \mapsto s_1^{h_1}] \circ \sigma')$, we must consider any type variable α such that:
- if $\alpha \neq \alpha_1$ and $\sigma'\alpha \neq \alpha_1$: then $(\phi \circ \sigma)\alpha = (\phi \circ \sigma')([\alpha_1 \mapsto s_1^{h_1}]\alpha) = (\phi \circ \sigma')\alpha = \delta\alpha$,
- if $\alpha \neq \alpha_1$ and $\sigma'\alpha = \alpha_1$: then $(\phi \circ \sigma)\alpha = (\phi \circ \sigma)([\alpha_1 \mapsto s_1^{h_1}]\alpha) = (\phi \circ \sigma')s_1^{h_1} = \delta s_1^{h_1} =_t \delta\alpha$,
- if $\alpha = \alpha_1$: then $(\phi \circ \sigma)\alpha = (\phi \circ \sigma)([\alpha_1 \mapsto s_1^{h_1}]\alpha) = (\phi \circ \sigma')s_1^{h_1} = \delta s_1^{h_1} =_t \delta\alpha$.

Thus, $\delta\alpha =_t (\phi \circ \sigma)\alpha$ for all variables α, that is, $\delta =_t \phi \circ \sigma$. Thus $\sigma \lesssim \delta$ as required.

□

3.4 Implementation and integration

The purpose of the whole Tom system is to take a Tom file composed of a mix between Java and Tom constructs as input and transforms it into a Java file as output. The system has a pipeline structure where each module process the given input and passes the result to the next one. All modules are written in Tom language which is bootstrapped. This allows the use of pattern matching and strategies in the development of the Tom system. In Figure 3.15 we illustrate the order of modules which are responsible for different compilation phases:

Parser During parsing, an Abstract Syntax Tree (AST) is produced with special nodes for Tom constructs such as **%match** and ‘ (backquote) constructs. These nodes are Gom terms, since the Tom grammar is defined by a Gom algebraic signature. Furthermore, in case a Gom algebraic signature as (part of) the input file, the parser calls the Gom tool to produce its mapping and respective Java implementation,

Syntax checker A series of verifications on the received AST is performed at the syntax checking phase. For instance, the verification that each function symbol found was declared or that there are no circular dependencies between matching conditions,

Desugarer The desugarer fills the type mapping with constructor and destructor Java methods for each Tom constructor. This phase also simplifies the AST by transforming all different kinds of nodes representing different host language constructors into a generic node. Moreover, each anonymous variable is named with a fresh name which means a name that was never used before into the code,

Typer The typer performs type inference considering types corresponding to the mappings of algebraic data types. The inferred types are propagated into the AST,

Type checker This module consists in checking the types/ranks of the symbols. For instance, it checks that two occurrences of the same variables have the same type,

Expander The expander transforms the AST in order to prepare the compilation. For instance, it generates constructs to allow the use of strategies, i.e. classes to traverse Tom terms,

Compiler The compiler locates a **%match** construct occurring in the AST and turns a rewrite rule composed of a list of several patterns into a list of several rewrite rules with a single pattern. These transformations are expressed in an intermediate language,

Optimizer This module is responsible for optimizing the code written in intermediate language. For instance, it reduces the number of assignments and tests corresponding to a **%match** construct,

Backend The backend consists in the code generation phase. During this phase, the constructs of the intermediate language are translated into host language instructions.

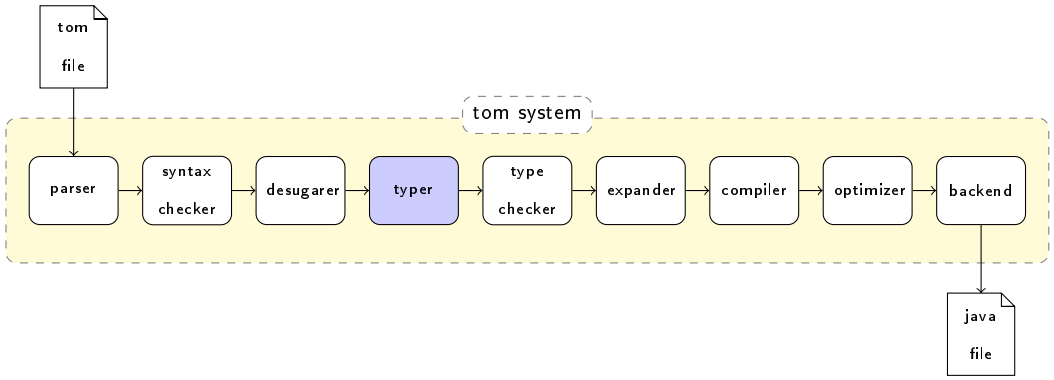


Figure 3.15: Modules of the Tom system.

The module *typer* corresponds to the implementation of the constraint-based type system presented in Section 3.3.4. Bringing theory to practice, the signature $\mathcal{F} \cup \mathcal{F}^\star$ and the set of base sorts are obtained during the parsing of the algebraic data types declared by the user. The base sorts correspond to ground types with decoration ?. All pairs (operator,rank) are collected together in a symbol table and all sorts are stored on a sort table. Both tables are available for all compilation phases. The symbol table works as the part of the context Γ referred to function symbols.

In addition to the rank, the symbol table also keeps other annotations relative to the construction and destruction of terms built from a given operator.

These annotations are useful when deciding if an operator is syntactic or variadic. Based on the annotations, the typing derivation of a term $\mathsf{g}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$ is obtained by the application of one of the typing rules [CT-Fun], [CT-Merge] and [CT-Elem] to the term. Simultaneously, in case g is a variadic operator, its rank $s_1^{?\star} \to s^?$ is replaced by $s_1^{?\star} \to s^g$. Then, the updated codomain is considered in the generation of type constraints.

After the syntax checking and desugaring phases, the typer is activated and starts by annotating each variable occurring in the AST with a fresh type variable. In the absence of the optional declaration of sort $|\tau|$ for a **%match** construct, τ is initially instantiated with a type variable. The type inference rules are applied each time a node representing a **%match** construct is found. For this purpose, the application of the typing rule [CT-Block] to an AST is controlled by the TopDownStopOnSuccess strategy which performs type inference in a top-down way and does not go under a node when [CT-Block] applies:

$$(\mathcal{T}opdown\mathcal{SOS}\ \mathsf{inference})(e) = (\mathcal{C}hoice\ \mathsf{inference}\ (\mathcal{A}ll\ (\mathcal{T}opdown\mathcal{SOS}\ \mathsf{inference})))(e)$$

This avoids the type inference of nested blocks in a separated scope. Accordingly to the type inference rules, instead of performing the rename of variables with same name but occurring in different rules $cond \longrightarrow action$, the typer creates *local contexts* Γ_i for each $rule_i$. All variables occurring in patterns of matching conditions and in numeric conditions of $rule_i$ and their annotations are inserted in Γ_i as a pair (variable,type variable). Thus, the union $\Gamma \cup \Gamma_i$ comprises all annotated variables and operators found in the node representing the $rule_i$, as required by the premises of the typing rule [CT-Block].

The type constraints collected during type inference are stored on a set $\mathcal{C}$. The insertion of a constraint c in $\mathcal{C}$ is done only if c is not in $\mathcal{C}$. Moreover, it keeps the constraint set in canonical form (see Definition 3.30) as ensured by the following rules:

$$\mathsf{insert}((\alpha =_t s^?), \mathcal{C}) \quad \Rightarrow \quad \begin{array}{ll} \mathcal{C} & \text{if } \alpha =_t s^v \in \mathcal{C} \vee s^v =_t \alpha \in \mathcal{C} \\ \{\alpha =_t s^?\} \cup \mathcal{C} & \text{otherwise} \end{array}$$

where $\alpha \in \mathcal{X}$, $s^? \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\})$ and $v \in \mathcal{F}^\star$.

During type inference, at the end of a block, the algorithm solveEqConstraints is called to loop over $\mathcal{C}$ and generate a substitution which is implemented by a hash map named Substitution. The following pseudocode ensures that the addition of a pair (Key,Value) keeps Substitution in saturated form:

```
GET Key and Value of pair to be added to Substitution
IF Value is in domain of Substitution THEN
    Add pair (Key,Substitution[Value]) to Substitution
ELSE
    Add pair (Key,Value) to Substitution
ENDIF

IF Key is in range of Substituion THEN
    FOR each CurrentKey in domain of Substitution
```

```
        IF Substitution[CurrentKey] is equal to Key THEN
            Overwrite pair (CurrentKey,Substitution[Key]) in Substitution
        ENDIF
    ENDLOOP
ENDIF
```

For the sake of readability, the implementation of the auxiliary algorithm isEq is slightly different from the description of Figure 3.14, since *err* is not a set of pairs (type,type) but a boolean variable. In case a type error is detected, the algorithm outputs an error message and sets *err* to true. Otherwise, *err* is assigned to false. The algorithm isEq returns *err* to solveEqConstraints as a flag variable. At the end of $\mathcal{C}$, the value of *err* determines whether the constraint set is unifiable. In the affirmative case, the resulting substitution is applied to the AST node for which the type inference was performed. Finally, at the end of the AST input, the typer replaces possible remaining type variables by an internal type EmptyType stating that it could not be inferred.

3.5 Synthesis

In this chapter we presented a type system for the Tom language, in particular Tom constructs for pattern matching. The system is composed of a type checking system and a constraint-based type inference system. Since Tom also implements associative pattern matching over variadic operators, i.e. list-matching, we were interested in defining both a way to distinguish different variadic operators and checking and inferring their types. For this purpose we formally defined the Tom syntax as well as the set of types composed of sorts decorated with function symbols. After the classification of terms by the use of types, we described an approximation of the operational semantics of the Tom core abstract syntax. This was useful to prove the safety of the type system.

In contrast to the declarative approach of the type checking system, the type inference system constitutes an algorithmic approach to perform static typing. We introduced constraints limiting types that expressions can have in typing judgments. Then, we defined rules to solve these constraints by unification. The whole type inference system was implemented and integrated in the Tom compiler as the *typer* module. It is available in the previous Tom version 2.8 available at the Tom source repository [6]. In addition to the test cases, the Tom compiler was successfully bootstrapped using the typer described in this chapter. In Chapter 4 we present an extension of the type system with subtyping which requires new algorithms for simplification and resolution of subtyping constraints.

6. Reference available at https://gforge.inria.fr/projects/tom/ . Visited on February 2012.

Chapter 4

Typing with subtypes in Tom

Although being proved sound and complete, the type system presented in Chapter 3 considers the equality of types as the only notion of type compatibility. In order to enhance the expressiveness of the pattern matching provided by Tom, we intend to allow for type inclusion at the pattern level. Therefore, this chapter introduces *subtyping* as a partial order over types and extends both the syntax of the Tom language and the semantics of typing judgments using subtypes. The type system proposed for Tom was first published in [KMT09] and later developed in a research report [Tav11].

4.1 Motivation

Since we focus on the definition of a type system for the Tom language without lost of compatibility with Java, we expect to handle essential features of this object-oriented host language such as inheritance. With subtyping, objects seen as algebraic terms can be of different subtypes but still uniformly manipulated as if belonging to a common supertype. This leads to a more expressive type system able to accept well-behaved programs which were previously considered as ill-typed.

When combined with subtyping, type checking rules are refined to allow a more permissive type compatibility. In this sense, types of variables and function symbols occurring in a term do not need to match exactly the type of its type signature anymore, but instead only be subtypes of it.

For the purpose of filling missing type annotations, the type inference system no longer generates a set of equality constraints to be solved. Instead, it generates a set of equality and subtyping (i.e. inequality) constraints to be simplified and subsequently solved.

To be coherent with Java, we opt for the use of nominal subtyping which requires subtyping relations between types to be explicitly declared. These relations are trivially checked by the type system each time an algebraic type declaration is found in order to provide a rich context for typing judgments.

4.2 Related Works

Generalizations of the Hindley/Milner system with subtyping have been proposed by several authors, such as Luca Cardelli [Car88] and John Mitchell [Mit84]. Cardelli proposed a nominal type system considering subtyping as a partial order over types and using it to model (multiple) inheritance between objects. Differently, Mitchell considered structural subtyping represented by coercion sets consisting only of constants and type variables and suggested the addition of automatic coercions to the well-known programming language ML [GMW79] based on type inference. He introduced the idea that type inference is a form of type checking and included subtyping constraints inside typing judgments.

Subtyping-constraint-based types systems has been thorough studied. Based on the work of Mitchell, You-Chin Fuh and Prateek Mishra [FM88] took interest in compute sets of coercions between base types and determine whether this set is solvable. Still in the presence of structural subtyping, Stefan Kaes [Kae92] was the first author to describe a type system incorporating parametric polymorphism, overloading, implicit coercions and recursive types. He defined a generic inference system to be instantiated, which is separated from the processing of the related constraint set. Correspondingly, Valery Trifonov and Scott Smith [TS96] also proposed an approach to separate the "syntactic aspects" of type inference from the details concerning constraint resolution.

In order to present a type system usable in practice, some authors addressed the problem of simplifying (large) constraint sets. Alexander Aiken and Edward Wimmers [AW92] proposed a powerful but complex inference algorithm which performs transformation of systems of constraints while preserving the set of solutions. It uses a dedicated formalism where types are connected by the union and intersection operators and organized into a type lattice. A simpler approach to simplifying constraints was given by François Pottier [Pot01]. As Trifonov and Smith, he also separated the type inference algorithm from the resolution of constraint sets using the notion of *constraint entailment*, i.e. constraint simplification which is solution preserving. Pottier's system uses a subsumption based on the constraint entailment and has inspired our constraint resolution algorithm. The main difference probably lies in the definition of types and their organization. Contrarily to Pottier's system, our type system does not support recursive types and our types are not organized in a type lattice and consequently his theorem about satisfiability of constraint sets does not hold for our type system. Moreover, Pottier was only interested in producing a smaller and simplified constraint set. In addition to constraint simplification, we in our turn focus on the generation of a single solution compatible with the one expected by the type system of the host language.

As regards type systems for order-sorted algebras, some approaches have been proposed as in the work of Clauss Hintermeier, Claude Kirchner and Hélène Kirchner [HKK98]. They present a type system for type constraints in membership equational logic where semantic declarations are also considered in addition to syntactic definitions. This allows enriching type annotations of

terms in order to perform type checking. It defines subtyping as a partial order over types that is interpreted as set inclusion, as proposed by Mitchell. Correspondingly, Jean-Pierre Jouannaud et al. [JKKM92] also described a nominal type system with subtyping relation as set inclusion providing parametric polymorphism, overloading and multiple inheritance. In the following, we present the Maude functional language which is based in the Jouannaud's type system. Then, we extend our interest to type inference for a combination of functional and object-oriented paradigms characterizing the Scala language.

4.2.1 Maude

As presented in Section 3.2.1, Maude [CDE+07] is a term rewriting language supporting features of OBJ [GWM+93] such as sorts, subsorts, dependent types, overloading of operators and parameterized programming (also called *generic programming*). Its functional modules are theories in membership equational logic introduced by Adel Bouhoula, Jean-Pierre Jouannaud and José Meseguer [BJM00, Mes97], where sorts are grouped into equivalence classes called *kinds*. Furthermore, functional modules are assumed to be *preregular* modulo some equational theory, which means that, for a given operator, the codomain sort of all its overloaded instances having a same arity must belong to the same kind. This property ensures that terms built from overloaded operators have a least sort. In order to avoid ambiguous expressions, Maude requires that, for overloaded instances with a same arity, if their domain sorts are pairwise of the same kind, then their codomain sorts must likewise be of the same kind [7]. Besides, two operators with the same syntactic form must have the same equational attributes. In Maude, multiple subsort relations can be declared in order to define a partial order among the sorts. However, the user is the responsible for the absence of cycles. The ExamplePeano module defined in Section 3.2.1 can be extended considering subsort relation:

```
1  fmod EXTENDED-EXAMPLEPEANO is
2     sorts Nat NatList .
3     subsort Nat < NatList .
4     op zero : -> Nat .
5     op suc : Nat -> Nat .
6     op empty : -> NatList .
7     op concNat : NatList NatList -> NatList [assoc id: empty] .
8  endfm
```

The subsort declaration at line 3 allows the associative binary operator concNat to be used as a variadic operator even for means of evaluation:

```
Maude> red concNat(zero, concNat(suc(zero), suc(suc(zero)))) .
reduce in EXTENDED-EXAMPLEPEANO : concNat(zero, concNat(suc(zero), suc(suc(zero)))) .
```

7. This requirement can be relaxed by the use of explicit type annotations of terms occurring in an expression.

```
rewrites : 0 in 0ms cpu (1ms real) (0 rewrites/second)
result NatList: concNat(zero, suc(zero), suc(suc(zero)))
```

As a property of membership equational logic, *sort decreasingness* requires that a term in normal form (i.e. an irreducible term) must have the least sort possible among the sorts of all the terms equivalent to it modulo an equational theory. Thus, terms built from the operator peanoPlus which implements the sum of two terms of sort NatList and is defined by the equational axioms below can be evaluated to an irreducible term of sort Nat as indicated in the last result line:

```
op peanoPlus : NatList NatList -> NatList .
vars X Y : NatList .
eq peanoPlus(zero, X) = X .
eq peanoPlus(suc(Y), X) = suc(peanoPlus(Y, X)) .
```

```
Maude> red peanoPlus(zero,suc(peanoPlus(suc(zero),suc(suc(zero))))) .
reduce in EXTENDED-EXAMPLEPEANO : peanoPlus(zero, suc(peanoPlus(suc(zero), suc(suc(zero))))) .
rewrites : 3 in 0ms cpu (0ms real) (3000000 rewrites/second)
result Nat: suc(suc(suc(suc(zero))))
```

In Maude, the solution of a pattern matching modulo an equational theory must be *kind-preserving*, i.e. patterns and subjects must be of the same kind. Otherwise a parsing error is raised:

```
Maude> match N:NatList <=? zero .
match in EXTENDED-EXAMPLEPEANO : N:NatList <=? zero .
Decision time: 0ms cpu (0ms real)

Solution 1
N:NatList --> zero

Maude> match N:Int <=? zero .
Warning: <standard input>, line 4: didn't expect token zero:
N:Int <=? zero <---*HERE*
Warning: <standard input>, line 4: no parse for command.
```

Maude allows performing pattern matching between terms whose sorts belong to the same kind. Thus, in spite of Tom, the sorts of subjects can be subsorts of those of patterns, although matching problems can have no solutions.

4.2.2 Scala

Scala [Ode11, Cre06] is a strongly statically typed language (presented in Section 3.2.2) which unifies the object-oriented and functional paradigms. Everything is an object in Scala, even functions which can be manipulated as values. Algebraic data types are described by either classes or mixins called

traits which are involved in multiple inheritance. Type and data constructors correspond respectively to abstract and case classes. Consequently, algebraic data types correspond to case classes in a flat hierarchy where the super class is an abstract class, as illustrated in the following Scala code:

```
1  abstract class Nat
2  case class suc(n: Nat) extends Nat

4  abstract class ZNat extends Nat
5  case class zero() extends ZNat

7  abstract class NatList
8  case class concNat(m: List[Nat]) extends NatList
```

The use of extends in case classes indicates that their instances are at same time instances of their super classes. Case classes are normal classes whose constructor may build patterns. Scala provides syntactic pattern matching on linear terms as described by the operational semantics defined by Burak Emir in his thesis [Emi07]. Although exhaustive, differently of Tom, only the first pattern which matches the subject is considered.

The Scala type system performs local type inference based on local type information of expressions. It supports parameterized types which can be limited by upper and/or lower bounds. The set of types is arranged by the conformance relation <: making a lattice (see Definition 1.16): Any is the supertype of all types and hence the *lub* of any pair of types; Nothing is a subtype of all types and likewise the *glb* of any pair of types (see Definition 1.15 of *lub* and *glb*). When extending pattern matching with subtypes, initially, the local type inference for patterns constructs a set of subtyping constraints over type variables. Then, similarly to what happens in Tom type inference system, it tries to solve the constraint set in order to find an instance of the type of pattern that *conforms* to (i.e. is a subtype of) the type of the subject. Consequently, the following method peanoPlus is legal and encodes the Example 2.2 in Scala:

```
9  def peanoPlus(t1: Nat, t2: Nat): Nat = (t1,t2) match {
10   case (zero(),x) => { return zero() }
11   case (suc(y),x) => { return suc(peanoPlus(y,x)) }
12 }
```

For patterns of parameterized types, the type parameter inference takes into account not only type of subjects but also type variables of type parameters and their bounds. The subtype behavior of parameterized types is controlled by *variances*. *Variances* constitute an annotation mechanism to constraint the way the annotated type variable may appear in the type or class by which it is bound. The variance annotations indicate covariant and contravariant dependency of instances of parameterized types and are considered in the constraint set.

4.3 The type system

As previously defined in Section 3.3, the set of base sorts $\mathcal{S}$ considered by our type system corresponds to the set of Tom algebraic data types declared by the user. We refine $\mathcal{S}$ with the addition of a partial order $<:$ over sorts called *subtyping*. It is a binary relation on $\mathcal{S}$ that satisfies reflexivity, transitivity and antisymmetry. We note that $\mathcal{S}$ does not define a lattice (see Definition 1.16) as we do not have a *glb* of any two no comparable sorts since multiple inheritance is not allowed. Then we extend the notion of *subtyping* over the set of types $\mathcal{T}y$ composed of decorated sorts (see Definition 3.1). This leads to $\mathcal{T}y$ equipped with another partial order $<:_t$.

Definition 4.1 (Subtyping over decorated sorts)**.** *Let $s_1, s_2 \in \mathcal{S}$ be two sorts and $h_1, h_2 \in \mathcal{F} \cup \mathcal{F}^\star$. The* subtyping over decorated sorts*, denoted by $<:_t$, is defined as follows:*

1. $s_1^? <:_t s_2^? \Leftrightarrow s_1 <: s_2$
2. $s_1^{h_1} <:_t s_2^? \Leftrightarrow s_1 <: s_2$
3. $s_1^{h_1} =_t s_2^{h_2} \Leftrightarrow s_1 <: s_2 \wedge h_1 = h_2$

where $<:$ stands for the subtyping relation over sorts.

In our type system, types are interpreted as unsorted terms and subtyping is nominal. Thus, subtyping relations must be explicitly declared, although multiple inheritance and overloading of operators are forbidden. For instance, given the types $\mathsf{Neg}^?$, $\mathsf{ZNat}^?$, $\mathsf{Nat}^?$ and $\mathsf{Int}^?$, the type system accepts the declaration $\mathsf{Neg}^? <:_t \mathsf{Int}^? \wedge \mathsf{Nat}^? <:_t \mathsf{Int}^?$ but in that case refuses the declaration $\mathsf{ZNat}^? <:_t \mathsf{Neg}^? \wedge \mathsf{ZNat}^? <:_t \mathsf{Nat}^?$. Moreover, in Tom a given function symbol cannot be overloaded on two different codomain types. For instance, the function symbol suc cannot have both codomain types $\mathsf{Neg}^?$ and $\mathsf{Nat}^?$.

Considering the set $\mathcal{T}y$ of types, the partial order $<:_t$ over $\mathcal{T}y$, the sets $\mathcal{F}$ and $\mathcal{F}^\star$ of function symbols and a countably infinite set $\mathcal{V}$ of variables, Tom terms are hereinafter built from the order-sorted term algebra $\mathcal{T}(\mathcal{T}y, <:_t, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ (see Definition 1.28). The set of Tom ground terms is denoted by $\mathcal{T}(\mathcal{T}y, <:_t, \mathcal{F} \cup \mathcal{F}^\star)$.

As we have seen in Chapter 2, algebraic data types in Tom are implemented by concrete data types in Java and this correspondence is defined by a mapping. Thus, the declaration of subtyping relations must be done in both type constructors and Java classes implementing them. Let us show below how to do this on an extract of code which extends the handwritten mapping of Section 2.2.1. There, we define two new type constructors ZNat and Int whose data constructors are respectively zero and uminus:

```
1  static class JInt{ }
2  static class JNat extends JInt { }
3  static class JZNat extends JNat { }

5  static class Juminus extends JInt {
6    public JNat n;
```

```
7     public Juminus() { }
8     public Juminus(JNat n) { this.n = n; }
9   }

11  static class Jzero extends JNat {
12    public Jzero() {}
13  }

15  %typeterm Int {
16    implement    { JInt }
17    is_sort(s)   { (s instanceof JInt) }
18    equals(t1,t2) { (t1.equals(t2)) }
19  }

21  %typeterm Nat extends Int {
22    implement    { JNat }
23    is_sort(s)   { (s instanceof JNat) }
24    equals(t1,t2) { (t1.equals(t2)) }
25  }

27  %typeterm ZNat extends Nat {
28    implement    { JZNat }
29    is_sort(s)   { (s instanceof JZNat) }
30    equals(t1,t2) { (t1.equals(t2)) }
31  }

33  %op Int uminus(n:Nat) {
34    is_fsym(s)   { (s instanceof Juminus) }
35    get_slot(n,s) { ((Juminus)s).n }
36    make(t0)     { new Juminus(t0) }
37  }

39  %op ZNat zero() {
40    is_fsym(s) { (s instanceof Jzero) }
41    make()     { new Jzero() }
42  }
```

The relationship between types ZNat, Nat and Int is declared by the keyword **extends** such that ZNat is a subtype of Nat which is a subtype of Int: lines 2 and 3, for Java types; lines 21 and 27, for algebraic types. Thus, the type system interprets this information as $\mathsf{ZNat}^? <:_t \mathsf{Nat}^? \wedge \mathsf{Nat}^? <:_t \mathsf{Int}^?$. This time, the data constructor zero is redefined for ZNat (instead of Nat) at line 39 and both a Java class and a new data constructor uminus for Int are declared at lines 5 and 33, respectively. This new type hierarchy is considered by Tom when performing pattern matching in order to accept patterns whose types are the same or a subtype of those of subjects. For that reason, we note that the method peanoPlus of Example 2.2 still works even after the redefinition of the type of zero. We illustrate the use of pattern matching on order-sorted terms by means of an example considering patterns of types ZNat, Nat and Int.

Example 4.2. *Consider the previously defined mapping. The following Java method describes an algorithm to calculate the addition of two (possibly negative)*

integers.

```
43  public JInt add(JInt t1, JInt t2) {
44    %match {
45      zero()     << t1 && x              << t2     -> { return `x; }
46      y          << t1 && uminus(zero()) << t2     -> { return `y; }
47      suc(y)     << t1 && x              << Nat t2 -> { return `suc(peanoPlus(y,x)); }
48      suc(y)     << t1 && uminus(suc(x)) << t2     -> { return `add(y,uminus(x)); }
49      uminus(y)  << t1 && uminus(x)      << t2     -> { return `uminus(peanoPlus(y,x)); }
50      uminus(y)  << t1 && suc(x)         << t2     -> { return `add(suc(x),uminus(y)); }
51    }
52    return null;
53  }
```

We note that the type parameter of the matching condition at line 47 is necessary to allow x to be given as an argument of peanoPlus in the right-hand side of the rule.

In the following sections, we describe the evaluation of Tom expressions, especially matching conditions, in the presence of subtyping. Then, we present an improvement in both type checking and type inference systems composing our type system in order to handle nominal subtyping in Tom.

4.3.1 Operational semantics

Although the introduction of a partial order over types does not modify Tom expressions themselves, this has an impact over the evaluation of matching conditions since the **%match** construct is parameterized by a optional declaration of sort $|\tau|$. The description of the evaluation of pattern matching considering subtyping was inspired by the work of Delia Kesner [Kes91].

The inference rules for the big-step operational semantics of the Tom language were previously presented in Section 3.3.2. They still hold with subtyping, except for those describing the evaluation of match-equations with variables as patterns. Hence, Figure 4.1 refines the rules for which an ascertainment over types and subtypes of patterns and subjects is required.

$$\frac{x \in \mathcal{V} \quad \tau =_t \mathsf{typeof}([\,],x) \quad \mathsf{typeof}([\,],\dot{u}) <:_t \tau}{\rho \Vdash x \ll |\tau| \ \dot{u} \Downarrow (\{\rho \leftarrow \{x \mapsto \dot{u}\}\}, \mathsf{accept})} \text{ E-MSubVarAcc}$$

$$\frac{x \in \mathcal{V} \quad \tau \neq_t \mathsf{typeof}([\,],x) \vee \neg(\mathsf{typeof}([\,],\dot{u}) <:_t \tau)}{\rho \Vdash x \ll |\tau| \ \dot{u} \Downarrow (\{\rho\}, \mathsf{reject})} \text{ E-MSubVarRej}$$

where $x \in \mathcal{V}$ and $\dot{u} \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$.

Figure 4.1: Evaluation rules for variable patterns considering subtyping.

We recapitulate that an evaluation judgment for Tom expressions has the form $\rho \Vdash \mathsf{t} \Downarrow (\rho', \dot{v})$ or $\rho \Vdash \mathsf{t} \Downarrow (\{\rho'_1, \ldots, \rho'_n\}, \dot{v} \backslash \dot{u})$ where $\{\rho'_1, \ldots, \rho'_n\}$ represents a set of environments. More than verifying structural matching, the reduction relation $\Downarrow$ on expressions must verify that the types of instances of patterns are subtypes of those expected for these patterns. Accordingly, the evaluation of a variable pattern is described by rules [E-MSubVarAcc] and [E-MSubVarRej]. The pattern yields a substitution if the value of the subject is of a type which is subtype of the explicit type annotation τ. Otherwise, the substitution is rejected. However, we note that the evaluation of patterns composed uniquely of star variables remains unchanged. It happens because types of star variables are indubitably decorated with a variadic operator v and must be instantiated exclusively by lists of these same types and with v as head symbol, since operator overloading is forbidden.

4.3.2 Type checking

The introduction of subtyping enables static typing to reject less well-behaved programs by weakening the notion of type compatibility. In a nominal type system as ours, the context may be enriched with subtyping declarations allowing terms be typed with subtypes of the types indicated in their type annotations.

Preliminary definitions

We refine the notion of context Γ with a new kind of pairs (type,type) to represent subtyping relations, in addition to the pairs (variable,type) and (operator,rank) previously introduced in Definition 3.13.

Definition 4.3 (Context with subtypes)**.** *A* context *is defined by:*

$$\Gamma ::= \varnothing \mid \Gamma; s_1^{h_1} <:_t s_2^{h_2} \mid \Gamma; \mathsf{x} : \tau \mid \Gamma; \mathsf{f} : s_1^?, \ldots, s_n^? \to s^? \mid \Gamma; \mathsf{v} : (s_1^?)^\star \to s^v$$

where $s_1^{h_1}, s_2^{h_2} \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\})$.

We recall that the Java operators are represented by Tom operators whose ranks are also declared in Γ as well as pairs (Java variable,type). Besides, access to the context is possible by the use of the binary function typeof as in Definition 3.14. The application of the transitive closure of $<:_t$ to all subtyping declarations found in Γ generates Γ^*.

Type checking rules

Subtype relations over types are handled in typing judgments $\Gamma \vdash e : \tau$ by a subsumption rule [Sub] which tells that every term t of type s_1^h is also of type $s^?$ if $s_1^h <:_t s^?$. This rule is presented in Figure 4.2 that recapitulates the type checking rules of Section 3.3.3. We note that [Sub] covers the previously defined rule [Gen], since $s^v <:_t s^?$ for any sort s and $\mathsf{v} \in \mathcal{F}^\star$.

$$\frac{}{\Gamma(\mathsf{x} : s^h) \vdash \mathsf{x} : s^h}\ \text{T-Var} \qquad \frac{}{\Gamma(\mathsf{x} : s^v) \vdash \mathsf{x*} : s^v}\ \text{T-SVar}$$

$$\frac{\Gamma \vdash e : s_1^h}{\Gamma(s_1^h <:_t s^?) \vdash e : s^?}\ \textbf{Sub} \qquad \frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma(\mathsf{x} : s^h) \vdash \mathsf{x@t} : s^h}\ \text{T-Alias}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : s_1^? \quad \dots \quad \Gamma \vdash \mathsf{t}_n : s_n^?}{\Gamma(\mathsf{f} : s_1^?, \dots, s_n^? \to s^?) \vdash \mathsf{f}(\mathsf{t}_1,\dots,\mathsf{t}_n) : s^?}\ \text{T-Fun} \qquad \frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma \vdash \mathsf{!t} : s^h}\ \text{T-Anti}$$

$$\frac{}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \mathsf{v}() : s^v}\ \text{T-Empty}$$

$$\frac{\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_{n-1}) : s^v \qquad \Gamma \vdash \mathsf{t}_n : s_1^?}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_n) : s^v}\ \text{T-Elem}$$
$$\text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v \text{ and } \mathsf{t}_n \neq \mathsf{x*}$$

$$\frac{\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_{n-1}) : s^v \qquad \Gamma \vdash \mathsf{t}_n : s^v}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_n) : s^v}\ \text{T-Merge}$$
$$\text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v \text{ or } \mathsf{t}_n = \mathsf{x*}$$

$$\frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma(\mathsf{x} : s^h) \vdash \mathsf{x} := \text{`}\mathsf{t} : wt}\ \text{T-Ass} \qquad \frac{\Gamma \vdash \mathsf{t} : s^h}{\Gamma \vdash \text{`}\mathsf{t} : wt}\ \text{T-BqTerm}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : s^? \qquad \Gamma \vdash \mathsf{t}_2 : s^?}{\Gamma \vdash (\mathsf{t}_1 \ll |s^h| \ \mathsf{t}_2) : wt}\ \text{T-Match} \qquad \frac{\Gamma \vdash \mathsf{t}_1 : s^? \qquad \Gamma \vdash \mathsf{t}_2 : s^?}{\Gamma \vdash (\mathsf{t}_1 \diamond \mathsf{t}_2) : wt}\ \text{T-Num}$$
$$\text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_i) = s^{h_i} \text{ for } i \in [1,2]$$

$$\frac{\Gamma \vdash (cond_1) : wt \qquad \Gamma \vdash (cond_2) : wt}{\Gamma \vdash (cond_1 \wedge cond_2) : wt}\ \text{T-Conj}$$

$$\frac{\Gamma \vdash (cond_1) : wt \qquad \Gamma \vdash (cond_2) : wt}{\Gamma \vdash (cond_1 \vee cond_2) : wt}\ \text{T-Disj}$$

$$\frac{\Gamma \vdash action_1 : wt \quad \dots \quad \Gamma \vdash action_n : wt}{\Gamma \vdash (action_1;\dots;action_n) : wt}\ \text{T-Action}$$

$$\frac{\Gamma \vdash cond : wt \qquad \Gamma \vdash action : wt}{\Gamma \vdash (cond \longrightarrow action) : wt}\ \text{T-Rule}$$

$$\frac{\Gamma \vdash rule_1 : wt \quad \dots \quad \Gamma \vdash rule_n : wt}{\Gamma \vdash \{rule_1;\dots;rule_n\} : wt}\ \text{T-Block}$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$, $\mathsf{v} \in \mathcal{F}^\star$ and $h \in \mathcal{F}^\star \cup \{?\}$

Figure 4.2: Type checking rules of Figure 3.7 where [Gen] is replaced by [Sub].

The type checking algorithm reads derivations bottom-up. Since the rule [Sub] can be applied to any kind of term, we consider a strategy where it is applied if and only if no other typing rule can be applied. In practice, [Sub] is combined with [T-Anti], [T-Alias], [T-Fun], [T-Elem] and [T-Merge] and the type s^h appearing in its premise is defined according to the result of function $\mathsf{typeof}(\Gamma,e)$. The typing rules are exhaustively applied until each branch of derivation reaches rules with no premises, or can make no further progress. The former case ensures that the original expression is well-typed while the latter one indicates a type error.

Example 4.4. *Let* $\Gamma_{init} = (ZNat^? <:_t Nat^?; \mathsf{zero} : ZNat^?; \mathsf{concNat} : (Nat^?)^\star \to NatList^{concNat}; \mathsf{x} : NatList^{concNat}; \mathsf{z} : NatList^{concNat}; \mathsf{pnat} : Nat^?; \mathsf{nList} : NatList^{concNat})$. *Let* $\Gamma = \Gamma^*_{init}$. *Consider the expression*

$$\{\mathsf{concNat(x*,pnat@!zero(),z*)} \ll |NatList^?|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}$$

which uses subtyping and is extensionally equivalent to the expression in Example 3.6

$$\{\mathsf{concNat(x*,pnat@suc(y),z*)} \ll |NatList^?|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}$$

It is well-typed and its deduction tree is given in Figure 4.3

Properties of type checking

Having extended our type system with subtyping, we want to verify that it maintains safety, i.e. that both progress and preservation theorems introduced in previous chapter still hold with subtypes. Therefore, in addition to the proofs done for the case without subtyping, we must check that the new type checking rule [Sub] satisfies these theorems. The inversion lemma is still useful for the development of these proofs but it must also be extended to consider the subsumption rule. Bear in mind that the overloading of operators are forbidden in our type system.

Lemma 4.5 (Inversion of the typing judgment)**.** *Let* x *be a variable and* f, v *be operators such that* $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$ *and* $\mathsf{v} \in \mathcal{F}^\star$. *Let* $\tau \in \mathcal{T}y \backslash \mathcal{X}$. *Let* $h, h_i \in \mathcal{F}^\star \cup \{?\}$ *and* t, $\mathsf{t}_i \in \mathcal{T}(\mathcal{T}y, <:_t, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *for* $i \in [1,n]$.

1. *If* $\Gamma \vdash \mathsf{x} : \tau$, *then:*
   (a) *either there is a type* s^h *such that* $\tau = s^h$ *with* $\mathsf{x} : s^h \in \Gamma$,
   (b) *or there are some types* $s^?$ *and* $s_1^{h_1}$ *such that* $\tau = s^?$ *with* $\Gamma \vdash \mathsf{x} : s_1^{h_1}$ *and* $s_1^{h_1} <:_t s^?$.
2. *If* $\Gamma \vdash \mathsf{x*} : \tau$, *then:*
   (a) *either there is a type* s^v *such that* $\tau = s^v$ *with* $\mathsf{x} : s^v \in \Gamma$,
   (b) *or there are some types* $s^?$ *and* s_1^v *such that* $\tau = s^?$ *with* $\Gamma \vdash \mathsf{x*} : s_1^v$ *and* $s_1^v <:_t s^?$ *for some* $\mathsf{v} \in \mathcal{F}^\star$.

3. *If* $\Gamma \vdash \mathsf{!t} : \tau$, *then there is a type* s^h *such that* $\tau = s^h$ *with* $\Gamma \vdash \mathsf{t} : s^h$.
4. *If* $\Gamma \vdash \mathsf{x@t} : \tau$, *then there is a type* s^h *such that* $\mathsf{x} : s^h \in \Gamma$ *and* $\Gamma \vdash \mathsf{t} : s^h$.
5. *If* $\Gamma \vdash \mathsf{f(t_1,\ldots,t_n)} : \tau$, *then:*
   (a) *either there are some types* $s^?$ *and* $s_i^?$ *such that* $\tau = s^?$ *with* $\mathsf{f} : s_1^?, \ldots, s_n^? \to s^? \in \Gamma$ *and* $\Gamma \vdash \mathsf{t}_i : s_i^?$ *for* $i \in [1, n]$,
   (b) *or there are some types* $s^?$ *and* $s_1^?$ *such that* $\tau = s^?$ *with* $\Gamma \vdash \mathsf{f(t_1,\ldots,t_n)} : s_1^?$ *and* $s_1^? <:_t s^?$.
6. *If* $\Gamma \vdash \mathsf{v()} : \tau$, *then:*
   (a) *either there are some types* $s_1^?$ *and* s^v *such that* $\tau = s^v$ *with* $\mathsf{v} : (s_1^?)^\star \to s^v \in \Gamma$,
   (b) *or there are some types* $s^?$ *and* s_1^v *such that* $\tau = s^?$ *with* $\Gamma \vdash \mathsf{v()} : s_1^v$ *and* $s_1^v <:_t s^?$.
7. *If* $\Gamma \vdash \mathsf{v(t_1,\ldots,t_n)} : \tau$, *then:*
   (a) *there are some types* $s_1^?$ *and* s^v *such that* $\tau = s^v$ *with* $\mathsf{v} : (s_1^?)^\star \to s^v \in \Gamma$ *and* $\Gamma \vdash \mathsf{v(t_1,\ldots,t_{n-1})} : s^v$ *and* $\Gamma \vdash \mathsf{t}_n : s^v$, *where* $\mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v$ *or* $t_n = \mathsf{x*}$,
   (b) *or there are some types* $s_1^?$ *and* s^v *such that* $\tau = s^v$ *with* $\mathsf{v} : (s_1^?)^\star \to s^v \in \Gamma$ *and* $\Gamma \vdash \mathsf{v(t_1,\ldots,t_{n-1})} : s^v$ *and* $\Gamma \vdash \mathsf{t}_n : s_1^?$, *where* $\mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v$ *and* $t_n \neq \mathsf{x*}$,
   (c) *or there are some types* $s^?$ *and* s_1^v *such that* $\tau = s^?$ *with* $\Gamma \vdash \mathsf{v(t_1,\ldots,t_n)} : s_1^v$ *and* $s_1^v <:_t s^?$.
8. *If* $\Gamma \vdash \mathsf{x} := \text{`}\mathsf{t} : \tau$ *then* $\tau = wt$ *and there is a type* s^h *such that* $\mathsf{x} : s^h \in \Gamma$ *and* $\Gamma \vdash \mathsf{t} : s^h$.
9. *If* $\Gamma \vdash \text{`}\mathsf{t} : \tau$, *then* $\tau = wt$ *and there is a type* $s^h$ *such that* $\Gamma \vdash \text{`}\mathsf{t} : s^h$.
10. *If* $\Gamma \vdash (\mathsf{t}_1 \ll |s^h| \ \mathsf{t}_2) : \tau$, *then* $\tau = wt$ *with* $\Gamma \vdash \mathsf{t}_1 : s^?$ *and* $\Gamma \vdash \mathsf{t}_2 : s^?$.
11. *If* $\Gamma \vdash (\mathsf{t}_1 \diamond \mathsf{t}_2) : \tau$, *then* $\tau = wt$ *and there is a type* $s^?$ *such that* $\Gamma \vdash \mathsf{t}_1 : s^?$ *and* $\Gamma \vdash \mathsf{t}_2 : s^?$.
12. *If* $\Gamma \vdash (cond_1 \wedge cond_2) : \tau$, *then* $\tau = wt$, $\Gamma \vdash (cond_1) : wt$ *and* $\Gamma \vdash (cond_2) : wt$.
13. *If* $\Gamma \vdash (cond_1 \vee cond_2) : \tau$, *then* $\tau = wt$, $\Gamma \vdash (cond_1) : wt$ *and* $\Gamma \vdash (cond_2) : wt$.
14. *If* $\Gamma \vdash (\mathsf{t}_1;\ldots;\mathsf{t}_n) : \tau$, *then* $\tau = wt$ *and* $\Gamma \vdash \tau_i : wt$ *for* $i \in [1, n]$.
15. *If* $\Gamma \vdash (cond \longrightarrow action) : \tau$, *then* $\tau = wt$ *and* $\Gamma \vdash cond : wt$ *and* $\Gamma \vdash action : wt$.
16. *If* $\Gamma \vdash (rule_1, \ldots, rule_n) : \tau$, *then* $\tau = wt$ *and* $\Gamma \vdash rule_i : wt$ *for* $i \in [1, n]$.

Proof. Immediate from the instance of the conclusion of the appropriate type checking rule combined with the rule [Sub] in Figure 4.2. Furthermore, the rule [Sub] must be considered for clauses 1b, 2b, 6b and 7c. □

As a first step to prove type safety, we show in the following that typing with subtyping is correct with respect to the operational semantics presented in Section 3.3.2.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{concNat}(): NatList^{concNat}}\ \text{T-Empty} \quad \dfrac{}{\Gamma \vdash \mathsf{x*}: NatList^{concNat}}\ \text{T-SVar}}{\Gamma \vdash \mathsf{concNat(x*)}: NatList^{concNat}}\ \text{T-Merge} \quad \vdots \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{zero}(): ZNat^{?}}\ \text{T-Fun}}{\Gamma \vdash !\mathsf{zero}(): ZNat^{?}}\ \text{T-Anti}}{\Gamma \vdash \mathsf{pnat@!zero}(): ZNat^{?}}\ \text{T-Alias}}{\Gamma \vdash \mathsf{pnat@!zero}(): Nat^{?}}\ \text{Sub}}{\Gamma \vdash \mathsf{concNat(x*,pnat@!zero())}: NatList^{concNat}}\ \text{T-Elem} \quad \vdots \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{z*}: NatList^{concNat}}\ \text{T-SVar}}{\Gamma \vdash \mathsf{concNat(x*,pnat@!zero(),z*)}: NatList^{concNat}}\ \text{T-Merge}}{\Gamma \vdash \mathsf{concNat(x*,pnat@!zero(),z*)}: NatList^{?}}\ \text{Sub} \quad \vdots \quad \dfrac{\dfrac{}{\Gamma \vdash \mathsf{nList}: NatList^{concNat}}\ \text{T-Var}}{\Gamma \vdash \mathsf{nList}: NatList^{?}}\ \text{Sub}}{\Gamma \vdash (\mathsf{concNat(x*,pnat@!zero(),z*)} \ll |NatList^{?}|\ \mathsf{nList}) : wt}\ \text{T-Match} \quad \vdots \quad \dfrac{\dfrac{}{\Gamma \vdash \mathsf{pnat}: ZNat^{?}}\ \text{T-Var}}{\Gamma \vdash \text{`}\mathsf{pnat}: wt}\ \text{T-BqTerm}}{\dfrac{\Gamma \vdash (\mathsf{concNat(x*,pnat@!zero(),z*)} \ll |NatList^{?}|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}) : wt}{\Gamma \vdash \{\mathsf{concNat(x*,pnat@!zero(),z*)} \ll |NatList^{?}|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\} : wt}\ \text{T-Block}}\ \text{T-Rule}$$

$$\Gamma = \Gamma^{*}_{init}$$
$$\Gamma_{init} = (ZNat^{?} <:_{t} Nat^{?};\ \mathsf{zero} : ZNat^{?};\ \mathsf{concNat} : (Nat^{?})^{\star} \to NatList^{concNat};$$
$$\mathsf{x} : NatList^{concNat};\ \mathsf{z} : NatList^{concNat};\ \mathsf{pnat} : Nat^{?};\ \mathsf{nList} : NatList^{concNat})$$

Figure 4.3: Example using the type checking algorithm with subtyping.

Theorem 4.6 (Progress). *Let e be a ground well-typed expression (that is, $\Gamma \vdash e : \tau$ for some $\tau \in \mathcal{T}y \backslash \mathcal{X}$ and $\mathcal{V}ar(e) = \emptyset$) such that e is not an anti-pattern. Then either e is a value or there is some value $e' \in \mathcal{V}al$ with either $\emptyset \Vdash e \Downarrow (\rho', e')$ or $\emptyset \Vdash e \Downarrow (\{\rho'_1, \ldots, \rho'_n\}, e')$.*

Proof. By induction on the derivation of $\vdash e : \tau$ like the proof of theorem 3.19. We just need to show that the new type checking rule [Sub] does not stop progress, which is trivial:

Case Sub: $e = \mathsf{t} \quad \mathsf{t} : s_1^{?}$

The result follows directly from the induction hypothesis. □

We introduce again the substitution lemma useful for the proof of type preservation.

Lemma 4.7 (Substitution). *If* $\Gamma(\mathsf{x} : s^h) \vdash e : \tau$, *such that* $\tau \in \mathcal{T}y \backslash \mathcal{X}$, *and* $\Gamma \vdash \mathsf{t} : s^h$, *then* $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{t}]e : \tau$.

Proof. Straightforward induction on a derivation of the statement $\Gamma(\mathsf{x} : s^h) \vdash e : \tau$. We consider a new case for type checking rule [Sub] whose proof is obtained directly form the induction hypothesis. The proof of the remaining cases is just like 3.20. □

To finish the proof of type safety, it remains to show that typing with subtyping is preserved by evaluation described by the operational semantics in Section 3.3.2. The statement of the preservation theorem is unchanged from the type checking without subtyping and its proof consider the type checking rules for constants of evaluation accept and reject presented in Figure 3.9.

Theorem 4.8 (Preservation). *If* $\Gamma \vdash e : \tau$ *and either* $\rho \Vdash e \Downarrow (\rho', e')$ *or* $\rho \Vdash e \Downarrow (\{\rho'_1, \ldots, \rho'_n\}, e')$ *such that* e *is not an anti-pattern and* $e' \in \mathcal{V}al$, *then* $\Gamma \vdash e' : \tau$, *for some* $\tau \in \mathcal{T}y \backslash \mathcal{X}$ *and* ρ, ρ'_i *with* $i \in [1, n]$.

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. Since the other cases were already tackled in the previous proof of theorem 3.21, we just need to deal with the new type checking rule [Sub]:

Case Sub: $e = \mathsf{t} \quad \tau = s^? \quad \Gamma \vdash \mathsf{t} : s_1^{h_1} \quad s_1^{h_1} <:_t s^?$

One of the rules [E-Var], [E-Val], [E-Fun] and [E-List] can be used to derive $\rho \Vdash e \Downarrow (\rho', e')$. By induction hypothesis, $\Gamma \vdash e' : s_1^{h_1}$. By application of rule [Sub], we obtain $\Gamma \vdash e' : s^?$ as required. □

4.3.3 Type inference

Besides requiring type annotations for all variables and operators, the type checking system presented in Section 4.3.2 needs rules to control its use in order to find the expected deduction tree of an expression. Without these rules it is possible to find more than one deduction tree for the same expression by application of [Sub]. Moreover, as a declarative typing mechanism, the type checking system requires that all terms be type annotated. Consequently, we are interested in defining another type inference system as an algorithmic way to calculate types of terms.

Preliminary definitions

When considering the set $\mathcal{T}y$ of types introduced in Definition 3.1, one can take advantage of type variables α to abstract types of terms in typing judgments. Type variables are instantiated by base types through the description of a type substitution and subsequent application of it to the typing context as defined in Section 3.3.4. This operation of substituting types for types variables must preserve the validity of typing judgments even in the presence of subtyping.

Theorem 4.9 (Preservation of typing under type substitution)**.** *Let σ be a type substitution. If $\Gamma \vdash e : \tau$ then $\sigma\Gamma \vdash \sigma e : \sigma\tau$.*

Proof. Straightforward induction on the derivation of the statement $\Gamma \vdash e : \tau$ like the proof of theorem 3.23. We just detail the case for the new type checking rule [Sub]:

$$\textit{Case Sub:}\quad \frac{e = \mathsf{t} \qquad \tau = \tau_1 \text{ with } \tau_2 <:_t \tau_1 \in \Gamma}{\Gamma \vdash \mathsf{t} : \tau_2}$$

Consider subcases where $\tau_1, \tau_2 \in \mathcal{D}om(\sigma)$ and $\tau_1, \tau_2 \notin \mathcal{D}om(\sigma)$. In all these cases, by induction hypothesis, $\sigma(\Gamma(\tau_2 <:_t \tau_1)) \vdash \sigma\mathsf{t} : \sigma\tau_2$ which states that $\sigma\tau_2 <:_t \sigma\tau_1 \in \sigma\Gamma$. By the application of [Sub], we have $\sigma(\Gamma(\tau_2 <:_t \tau_1)) \vdash \sigma\mathsf{t} : \sigma\tau_1$, as required. □

A type substitution constitutes a solution for typing judgment as established in Definition 3.24. The existence of this solution determines that the Tom expression in the given typing judgment is well-typed. A common approach to perform type inference with subtyping is based on limiting type variables through subtyping constraints in order to find a type substitution satisfying these constraints. In this sense, we must extend the notion of *constraint set* initially composed of equality constraints as introduced in Definition 3.25.

Definition 4.10 (Constraint set)**.** *Consider the set $\mathcal{T}y$ of types. A* constraint set *$\mathcal{C}$ is a set of constraints defined by the following algebraic grammar:*

$$c ::= \tau_1 =_t \tau_2 \mid \tau_1 <:_t \tau_2$$

where $c \in \mathcal{C}$, $\tau_1, \tau_2 \in \mathcal{T}y$.

A ground constraint *is a constraint with no variables, i.e. a constraint built from ground types.*

A substitution σ is said to satisfy *an equation $\tau_1 =_t \tau_2$ if $\sigma(\tau_1) =_t \sigma(\tau_2)$. Similarly, σ is said to* satisfy *a subtyping constraint $\tau_1 <:_t \tau_2$ if $\sigma(\tau_1) <:_t \sigma(\tau_2)$. Thus, σ satisfies $\mathcal{C}$ if it satisfies all constraints in $\mathcal{C}$. This is written $\sigma \models \mathcal{C}$.*

The set $\mathcal{V}ar(\mathcal{C})$ denotes the set of type variables existing in $\mathcal{C}$.

$$\frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Var} \qquad \frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x*} : \alpha) \vdash_{ct} \mathsf{x*} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-SVar}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash_{ct} !\mathsf{t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Anti} \qquad \frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \quad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x@t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Alias}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \quad \dots \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_n \bullet \mathcal{C}_n \qquad \mathcal{C} = \{\alpha_0 =_t s^?\} \bigcup_{i=1}^{n} \mathcal{C}_i \cup \{\alpha_i <:_t s_i^?\}}{\Gamma(\mathsf{f} : s_1^?, \dots, s_n^? \to s^?) \vdash_{ct} \mathsf{f}(\mathsf{t}_1,\dots,\mathsf{t}_n) : \alpha_0 \bullet \mathcal{C}}\ \text{CT-Fun}$$

$$\frac{\mathcal{C} = \{\alpha_1 =_t s^v\}}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash_{ct} \mathsf{v}() : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Empty}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v, \alpha_2 <:_t s_1^?\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \quad \text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v \text{ and } \mathsf{t}_n \neq \mathsf{x*}\ \text{CT-Elem}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_1 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\dots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \quad \text{if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v \text{ or } \mathsf{t}_n = \mathsf{x*}\ \text{CT-Merge}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \quad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} := \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-Ass} \qquad \frac{\Gamma \vdash \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-BqTerm}$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$, $\mathsf{v} \in \mathcal{F}^\star$ and α_i are fresh type variables for $i \in [1, n]$

Figure 4.4: Type inference rules considering subtyping.

Type inference rules

Equality and subtyping constraints are calculated according to the application of rules of type inference system presented in Figures 4.4 and 4.5 where constraint typing judgments have the form $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ as in Section 3.3.4.

In order to infer the type of a given expression e, the context Γ is initialized to:

1. subtyping relations of the form $s_1^? <:_t s_2^?$ where $s_1^?, s_2^? \in \mathcal{T}y \backslash (\mathcal{X} \cup \{wt\})$,
2. a pair of the form $(\mathsf{f} : s_1^?, \dots, s_n^? \to s^?)$ for each syntactic operator f occurring

in the typing judgment where $s_i^?, s^? \in \mathcal{T}y\backslash(\mathcal{X} \cup \{wt\})$ for $i \in [1,n]$,

3. a pair of the form $(\mathsf{v} : s_1^{?\star} \to s^v)$ for each variadic operator v occurring in the typing judgment where $s_1^?, s^v \in \mathcal{T}y\backslash(\mathcal{X} \cup \{wt\})$,
4. a pair of the form $(\mathsf{x} : \alpha)$ for each variable x and star variable $\mathsf{x*}$ occurring in the typing judgment where $\alpha \in \mathcal{X}$ is a fresh type variable.

Each type variable introduced into a sub-derivation is a fresh type variable and the fresh type variables in different sub-derivations are distinct. All the type inference rules presented in this section are identical to the ones in Section 3.3.4, except rules [CT-Fun], [CT-Elem], [CT-Match] and [CT-Num]. In the rule [CT-Fun], assuming that $s_1^?, \ldots, s_n^? \to s^?$ is the rank of f in Γ, a type α such that $\alpha <:_t s^?$ is a valid type for the expression $\mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$ and each term t_i occurring in it is of type α_i which must be a subtype of $s_i^?$ for $i \in [1,n]$. The insertion of an element into a list is held by rule [CT-Elem]: for a non-empty list $\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$, if the element t_n is of type of the domain of v and its head symbol is different from v, then types α_1 and α such that $\alpha_1 <:_t s_1^?$ and $\alpha =_t s^v$ are valid types for t_n and for the original list concatenated with t_n, respectively. The rule [CT-Match] is applied to a pattern matching condition whose construct is declared with sort $|\tau|$. It asserts that, for a well-typed matching condition, τ must be equal to the type of pattern modulo equality of decorated sort and also a subtype of the type of subject. However, for numeric conditions, the rule [CT-Num] tells that types of terms t_1 and t_2 must have a common lower bound which obliges both types to be comparable.

Example 4.11. *Let* $\Gamma_{init} = \Gamma' \cup \Gamma_1$ *such that* $\Gamma_1 = (\mathsf{x} : \alpha_2; \mathsf{z} : \alpha_3; \mathsf{pnat} : \alpha_4)$ *and* $\Gamma' = (\mathsf{zero} : ZNat^?; \mathsf{concNat} : (Nat^?)^\star \to NatList^{concNat}; \mathsf{nList} : \alpha_1)$. *Let* $\Gamma = \Gamma_{init}^*$. *Consider the expression*

$$\{\mathsf{concNat(x*,pnat@!zero(),z*)} \ll |NatList^?|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}$$

which uses subtyping and is extensionally equivalent to the expression in Example 3.6

$$\{\mathsf{concNat(x*,pnat@suc(y),z*)} \ll |NatList^?|\ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}$$

The latter expression can be deduced by the tree given in Figure 4.6, where a constraint set $\mathcal{C}$ *is generated.*

Properties of type inference

Having extended the notion of constraint set to encompass subtyping constraints, we must adapt the concept of *solution* for a typing judgment based on constraints.

Definition 4.12 (Solution for constraint typing judgment)**.** *Let* Γ *be a context and* e *a Tom expression. Suppose that* $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$. *A* solution *for* $(\Gamma, e, \tau, \mathcal{C})$ *is a pair* (σ, τ') *such that* σ *satisfies* $\mathcal{C}$ *and* $\sigma\tau <:_t \tau'$, *where* $\tau' \in \mathcal{T}y\backslash\mathcal{X}$.

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\tau =_t \alpha_1, \alpha_1 <:_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathsf{t}_1 \ll |\tau| \ \mathsf{t}_2) : wt \bullet \mathcal{C}} \text{ CT-Match}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha <:_t \alpha_1, \alpha <:_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathsf{t}_1 \diamond \mathsf{t}_2) : wt \bullet \mathcal{C}} \text{ CT-Num}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \wedge \ldots \wedge cond_n) : wt \bullet \mathcal{C}} \text{ CT-Conj}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \vee \ldots \vee cond_n) : wt \bullet \mathcal{C}} \text{ CT-Disj}$$

$$\frac{\Gamma \vdash_{ct} action_1 : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} action_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (action_1; \ldots; action_n) : wt \bullet \mathcal{C}} \text{ CT-Action}$$

$$\frac{\Gamma \vdash_{ct} (cond) : wt \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} (action) : wt \bullet \mathcal{C}_2 \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (cond \longrightarrow action) : wt \bullet \mathcal{C}} \text{ CT-Rule}$$

$$\frac{\Gamma \cup \Gamma_1 \vdash_{ct} rule_1 : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \cup \Gamma_n \vdash_{ct} rule_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \bigcup_{i=1}^{n} \Gamma_i \vdash_{ct} \{rule_1; \ldots; rule_n\} : wt \bullet \mathcal{C}} \text{ CT-Block}$$

where α_1, α_2 are fresh type variables

Figure 4.5: Type inference rules considering subtyping – continuation.

Since our type checking and type inference systems address the same issue, we shall show that every solution proposed by the inference rules in an algorithmic way also arises from the declarative approach held by the checking rules.

$$\frac{\mathcal{C}_{10} = \{\alpha_5 =_t NatList^{concNat}\}}{\Gamma \vdash_{ct} \textsf{concNat()}\colon \alpha_5 \bullet \mathcal{C}_{10}}\ \text{CT-EMPTY}$$

$$\frac{\mathcal{C}_{11} = \{\alpha_2 =_t \alpha_5\}}{\Gamma \vdash_{ct} \textsf{x*}\colon \alpha_5 \bullet \mathcal{C}_{11}}\ \text{CT-SVAR}$$

$$\frac{\mathcal{C}_8 = \{\alpha_5 =_t NatList^{concNat}\} \cup \mathcal{C}_{10} \cup \mathcal{C}_{11}}{\Gamma \vdash_{ct} \textsf{concNat(x*)}\colon \alpha_5 \bullet \mathcal{C}_8}\ \text{CT-MERGE}$$

$$\frac{\mathcal{C}_{13} = \{\alpha_7 =_t ZNat^?\}}{\Gamma \vdash_{ct} \textsf{zero()} : \alpha_7 \bullet \mathcal{C}_{13}}\ \text{CT-FUN}$$

$$\frac{\mathcal{C}_{12} = \mathcal{C}_{13}}{\Gamma \vdash_{ct} \textsf{!zero()}\colon \alpha_7 \bullet \mathcal{C}_{12}}\ \text{CT-ANTI}$$

$$\frac{\mathcal{C}_9 = \{\alpha_4 =_t \alpha_7\} \cup \mathcal{C}_{12}}{\Gamma \vdash_{ct} \textsf{pnat@!zero()}\colon \alpha_7 \bullet \mathcal{C}_9}\ \text{CT-ALIAS}$$

$$\frac{\mathcal{C}_6 = \{\alpha_5 =_t NatList^{concNat}, \alpha_7 <:_t Nat^?\} \cup \mathcal{C}_8 \cup \mathcal{C}_9}{\Gamma \vdash_{ct} \textsf{concNat(x*,pnat@!zero())}\colon \alpha_5 \bullet \mathcal{C}_6}\ \text{CT-ELEM}$$

$$\frac{\mathcal{C}_7 = \{\alpha_3 =_t \alpha_5\}}{\Gamma \vdash_{ct} \textsf{z*}\colon \alpha_5 \bullet \mathcal{C}_7}\ \text{CT-SVAR}$$

$$\frac{\mathcal{C}_4 = \{\alpha_5 =_t NatList^{concNat}\} \cup \mathcal{C}_6 \cup \mathcal{C}_7}{\Gamma \vdash_{ct} \textsf{concNat(x*,pnat@!zero(),z*)}\colon \alpha_5 \bullet \mathcal{C}_4}\ \text{CT-MERGE}$$

$$\frac{\mathcal{C}_5 = \{\alpha_1 =_t \alpha_6\}}{\Gamma \vdash_{ct} \textsf{nList}\colon \alpha_6 \bullet \mathcal{C}_5}\ \text{CT-VAR}$$

$$\frac{\mathcal{C}_2 = \{NatList^? =_t \alpha_5, \alpha_5 <:_t \alpha_6\} \cup \mathcal{C}_4 \cup \mathcal{C}_5}{\Gamma \vdash_{ct} \textsf{(concNat(x*,pnat@!zero(),z*)} \ll |NatList^?|\ \textsf{nList)} : wt \bullet \mathcal{C}_2}\ \text{CT-MATCH}$$

$$\frac{\mathcal{C}_{14} = \{\alpha_4 =_t \alpha_8\}}{\Gamma \vdash_{ct} \textsf{pnat}\colon \alpha_8 \bullet \mathcal{C}_{14}}\ \text{CT-VAR}$$

$$\frac{\mathcal{C}_3 = \mathcal{C}_{14}}{\Gamma \vdash_{ct} \textsf{`pnat}\colon wt \bullet \mathcal{C}_3}\ \text{CT-BQTERM}$$

$$\frac{\mathcal{C}_1 = \mathcal{C}_2 \cup \mathcal{C}_3}{\Gamma \vdash_{ct} \textsf{(concNat(x*,pnat@!zero(),z*)} \ll |NatList^?|\ \textsf{nList} \longrightarrow \textsf{`pnat)} : wt \bullet \mathcal{C}_1}\ \text{CT-RULE}$$

$$\frac{\mathcal{C} = \mathcal{C}_1}{\Gamma \vdash_{ct} \textsf{\{concNat(x*,pnat@!zero(),z*)} \ll |NatList^?|\ \textsf{nList} \longrightarrow \textsf{`pnat\}} : wt \bullet \mathcal{C}}\ \text{CT-BLOCK}$$

Figure 4.6: Example using the type inference algorithm with subtyping.

Theorem 4.13 (Soundness of constraint typing)**.** *Suppose that* $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ *is a valid sequent. If* (σ, τ') *is a solution for* $(\Gamma, e, \tau, \mathcal{C})$*, then it is also a solution for* (Γ, e) *(i.e. e is well-typed in* Γ*).*

Proof. By induction on the given constraint typing derivation for $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$. We just detail the most noteworthy cases of this proof which differs from the proof of 3.28.

Case CT-ELEM:
$$e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$$
$$\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_1 \bullet \mathcal{C}_2$$
$$\mathcal{C} = \{\alpha =_t s^v, \alpha_1 <:_t s_1^?\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$$

We are given that (σ, s^v) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e, \alpha, \mathcal{C})$, that is, σ satisfies $\mathcal{C}$ and $\sigma\alpha <:_t s^v$. Thus, $\sigma\alpha =_t s^v$, and $\sigma\alpha_1 <:_t s_1^?$. Since σ satisfies $\mathcal{C}_1$ and $\mathcal{C}_2$, $(\sigma, \sigma\alpha)$ and $(\sigma, \sigma\alpha_1)$ are solutions for $(\Gamma, \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}), \alpha, \mathcal{C}_1)$ and $(\Gamma, \mathsf{t}_n, \alpha_1, \mathcal{C}_2)$, respectively. By induction hypothesis, we have $\sigma\Gamma \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1})) : \sigma\alpha$ and $\sigma\Gamma \vdash \sigma\mathsf{t}_n : \sigma\alpha_1$. Since $\sigma\alpha <:_t s^v$, by [SUB] we obtain $\sigma\Gamma \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1})) : s^v$. Since $\sigma\alpha_1 <:_t s_1^?$, by [SUB] we obtain $\sigma\Gamma \vdash \mathsf{t}_n : s_1^?$. Then, by [T-ELEM] we obtain $\sigma(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v) \vdash \sigma(\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) : s^v$, that is, (σ, s^v) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e)$ as required.

Case CT-MATCH:
$$(\mathsf{t}_1 \ll |\tau|\ \mathsf{t}_2)$$
$$\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2$$
$$\mathcal{C} = \{\tau =_t \alpha_1, \tau <:_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$$

We are given that (σ, wt) is a solution for $(\Gamma, e, wt, \mathcal{C})$, that is, σ satisfies $\mathcal{C}$ and $\sigma wt <:_t wt$. Thus, $\sigma\tau =_t \sigma\alpha_1$ and $\sigma\tau <:_t \sigma\alpha_2$. Since σ satisfies $\mathcal{C}_1$ and $\mathcal{C}_2$, $(\sigma, \sigma\alpha_1)$ and $(\sigma, \sigma\alpha_2)$ are solutions for $(\Gamma, \mathsf{t}_1, \alpha_1, \mathcal{C}_1)$ and $(\Gamma, \mathsf{t}_2, \alpha_2, \mathcal{C}_2)$, respectively. By induction hypothesis, we have $\sigma\Gamma \vdash \sigma\mathsf{t}_1 : \sigma\alpha_1$ and $\sigma\Gamma \vdash \sigma\mathsf{t}_2 : \sigma\alpha_2$. Since $\sigma\tau <:_t \sigma\alpha_2$, by [SUB] we obtain $\sigma\Gamma \vdash \mathsf{t}_1 : \sigma\alpha_2$. By [T-MATCH] we obtain $\sigma\Gamma \vdash \sigma(\mathsf{t}_1 \ll |\sigma\alpha_2|\ \mathsf{t}_2) : wt$, that is, (σ, wt) is a solution for (Γ, e) as required.

Case CT-NUM:
$$(\mathsf{t}_1 \ll |\tau|\ \mathsf{t}_2)$$
$$\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2$$
$$\mathcal{C} = \{\alpha <:_t \alpha_1, \alpha <:_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$$

We are given that (σ, wt) is a solution for $(\Gamma, e, wt, \mathcal{C})$, that is, σ satisfies $\mathcal{C}$ and $\sigma wt <:_t wt$. Thus, $\sigma\alpha <:_t \sigma\alpha_1$ and $\sigma\alpha <:_t \sigma\alpha_2$. Since σ satisfies $\mathcal{C}_1$ and $\mathcal{C}_2$, $(\sigma, \sigma\alpha_1)$ and $(\sigma, \sigma\alpha_2)$ are solutions for $(\Gamma, \mathsf{t}_1, \alpha_1, \mathcal{C}_1)$ and $(\Gamma, \mathsf{t}_2, \alpha_2, \mathcal{C}_2)$, respectively. By induction hypothesis, we have $\sigma\Gamma \vdash \sigma\mathsf{t}_1 : \sigma\alpha_1$ and $\sigma\Gamma \vdash \sigma\mathsf{t}_2 : \sigma\alpha_2$. Since multiple inheritance is forbidden in the type system, thus $\sigma\alpha <:_t \sigma\alpha_1$ and $\sigma\alpha <:_t \sigma\alpha_2$ means that either $\sigma\alpha_1 <:_t \sigma\alpha_2$ or $\sigma\alpha_2 <:_t \sigma\alpha_1$. Proceed by cases:

1. Subcase $\sigma\alpha_1 <:_t \sigma\alpha_2$: since $\sigma\Gamma \vdash \sigma\mathsf{t}_1 : \sigma\alpha_1$, by [SUB] we obtain $\sigma\Gamma \vdash \sigma\mathsf{t}_1 : \sigma\alpha_2$. By [T-NUM] we obtain $\sigma\Gamma \vdash \sigma(\mathsf{t}_1 \ll |\tau|\ \mathsf{t}_2) : wt$ as required.
2. Subcase $\sigma\alpha_2 <:_t \sigma\alpha_1$: since $\sigma\Gamma \vdash \sigma\mathsf{t}_2 : \sigma\alpha_2$, by [SUB] we obtain $\sigma\Gamma \vdash \sigma\mathsf{t}_2 : \sigma\alpha_1$. By [T-NUM] we obtain $\sigma\Gamma \vdash \sigma(\mathsf{t}_1 \ll |\tau|\ \mathsf{t}_2) : wt$ as required.

□

We introduce again the statement of the completeness theorem 3.29 in order to show that a solution given by the checking rules can be extended to a solution proposed by the inference rules even in the presence of subtyping.

Theorem 4.14 (Completeness of constraint typing)**.** *Suppose that* $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ *is derived by* π*. Write* $\mathcal{V}ar(\pi)$ *for the set of all type variables appearing in* π *and write* $\sigma\backslash\mathcal{V}ar(\pi)$ *for the substitution that is the identity for all the variables in* $\mathcal{V}ar(\pi)$ *and otherwise behaves like* σ*. If* (σ, τ') *is a solution for* (Γ, e) *and* $\mathcal{D}om(\sigma) \cap \mathcal{V}ar(\pi) = \emptyset$*, then there is some solution* (σ', τ') *for* $(\Gamma, e, \tau, \mathcal{C})$ *such that* $\sigma'\backslash\mathcal{V}ar(\pi) = \sigma$*.*

Proof. By induction on the given constraint typing derivation, but we must take care with fresh names of variables. We just detail the most noteworthy cases of this proof which differs from the proof of 3.29.

Case CT-Elem: $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$
$\pi_1 = \Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha \bullet \mathcal{C}_1$ $\quad \pi_2 = \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_1 \bullet \mathcal{C}_2$
$\mathcal{C} = \{\alpha =_t s^v, \alpha_1 <:_t s_1^?\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$ $\quad \mathcal{V}ar(\pi) = \{\alpha, \alpha_1\}$
$\mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v$

We are given that (σ, s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e)$ and $\mathcal{D}om(\sigma) \cap \mathcal{V}ar(\pi) = \emptyset$. By the inversion lemma 4.5, clauses 7a, 7b, and 7c apply. We reason by subcases as follows:

1. Subcase [T-Merge]: if $h = v$, then there is a type $s_1^?$ such that $\mathsf{v} : (s_1^?)^\star \to s^v \in \sigma\Gamma$ with $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ and $\sigma\Gamma \vdash \mathsf{t}_n : s^v$. But, since $\mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v$, $\sigma\Gamma \vdash \mathsf{t}_n : s^v$ can not be derived even from [Sub]. Thus, it is not a relevant subcase.
2. Subcase [T-Elem]: if $h = v$, then there is a type $s_1^?$ such that $\mathsf{v} : (s_1^?)^\star \to s^v \in \sigma\Gamma$ with $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ and $\sigma\Gamma \vdash \mathsf{t}_n : s_1^?$. By induction hypothesis, there are solutions (σ_1, s^v) for $(\Gamma, \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}), \alpha, \mathcal{C}_1)$ and $(\sigma_2, s_1^?)$ for $(\Gamma, \mathsf{t}_n, \alpha_1, \mathcal{C}_2)$ such that $\mathcal{D}om(\sigma_1) \cap \mathcal{V}ar(\pi_1) = \emptyset = \mathcal{D}om(\sigma_2) \cap \mathcal{V}ar(\pi_2)$. Define $\sigma' = \sigma_1 \cup \sigma_2 \cup \sigma$. Thus σ' satisfies $\mathcal{C}$ and $\sigma'\alpha <:_t s^v$, that is, (σ', s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e, \alpha, \mathcal{C})$ as required.
3. Subcase [Sub]: assume that $s^h = s_2^?$, then there is a type s^v such that $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v$ and $s^v <:_t s_2^?$. Proceed as in subcase 2 and consequently define $\sigma' = \sigma_1 \cup \sigma_2 \cup \sigma$. Thus σ' satisfies $\mathcal{C}$ and $\sigma'\alpha <:_t s^v$. By transitivity, $\sigma'\alpha <:_t s_2^?$, that is, (σ', s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e, \alpha, \mathcal{C})$ as required.

Case CT-Merge: $e = \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$
$\pi = \Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha \bullet \mathcal{C}_1$ $\quad \pi = \Gamma \vdash_{ct} \mathsf{t}_n : \alpha \bullet \mathcal{C}_2$
$\mathcal{C} = \{\alpha =_t s^v\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$ $\quad \mathcal{V}ar(\pi) = \{\alpha\}$
$\mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v$

We are given that (σ, s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e)$. By the inversion lemma 4.5, clauses 7a, 7b and 7c apply, as in previous case. We reason by subcases as follows:

1. Subcase [T-MERGE]: if $h = v$, then there is a type $s_1^?$ such that $\mathsf{v} : (s_1^?)^\star \to s^v \in \sigma\Gamma$ with $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ and $\sigma\Gamma \vdash \mathsf{t}_n : s^v$. By induction hypothesis, there are solutions (σ_1, s^v) for $(\Gamma, \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}), \alpha, \mathcal{C}_1)$ and (σ_2, s^v) for $(\Gamma, \mathsf{t}_n, \alpha, \mathcal{C}_2)$ such that $\mathcal{D}om(\sigma_1) \cap \mathcal{V}ar(\pi_1) = \emptyset = \mathcal{D}om(\sigma_2) \cap \mathcal{V}ar(\pi_2)$. So, $\sigma_1\alpha <:_t s^v$ and $\sigma_2\alpha <:_t s^v$. Define $\sigma' = \{\alpha \mapsto s^v\} \cup \sigma_1 \cup \sigma_2 \cup \sigma$. Since σ_1 and σ_2 satisfy respectively $\mathcal{C}_1$ and $\mathcal{C}_2$, σ' also satisfies both $\mathcal{C}_1$ and $\mathcal{C}_2$, by transitivity of $<:_t$. Thus σ' satisfies $\mathcal{C}$ and $\sigma'\alpha <:_t s^v$, that is, (σ', s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e, \alpha, \mathcal{C})$ as required.
2. Subcase [T-ELEM]: if $h = v$, then there is a type $s_1^?$ such that $\mathsf{v} : (s_1^?)^\star \to s^v \in \sigma\Gamma$ with $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : s^v$ and $\sigma\Gamma \vdash \mathsf{t}_n : s_1^?$, where $\mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v$. But, since $\mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v$, this is not a relevant subcase.
3. Subcase [SUB]: assume that $s^h = s_2^?$, then there is a type s^v such that $\sigma\Gamma \vdash \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : s^v$ and $s^v <:_t s_2^?$. Proceed as in subcase 1 and consequently define $\sigma' = \{\alpha \mapsto s^v\} \cup \sigma_1 \cup \sigma_2 \cup \sigma$. Thus σ' satisfies $\mathcal{C}$ and $\sigma'\alpha <:_t s^v$. By transitivity, $\sigma'\alpha <:_t s_2^?$, that is, (σ', s^h) is a solution for $(\Gamma(\mathsf{v} : (s_1^?)^\star \to s^v), e, \alpha, \mathcal{C})$ as required.

□

4.3.4 Constraint resolution

The type inference rules describe an algorithmic way to map types of terms occurring in an expression to a constraint set. When considering subtyping, the constraint set obtained can be divided into two subsets: an equality constraint subset and a subtyping constraint subset. The former subset is solved by unification through algorithm solveEqConstraints explained in Section 3.3.5, generating a substitution set σ in saturated form (see Definition 3.32) for type variables. If none errors were found during the unification (i.e. $err = \emptyset$), then the substitution is applied to the subtyping constraint subset. Otherwise, errors are raised out and the constraint resolution stops. In this section we describe a simplification algorithm inspired by the work of François Pottier [Pot98b, Pot98a, Pot01] to propagate subtyping constraints. It is done by the combination of three simplification phases which keep σ in saturated form. The propagation of subtyping constraints is described in the following and afterwards we present a phase responsible for the generation of *solution*. The last phase is the one for garbage collection on possible remaining constraints. As in equality constraint resolution, the solution σ and the set err of errors are initially empty. Moreover, $\mathcal{C}$ has only equality or subtyping constraints between types τ_1 and τ_2. However, a non-empty set err indicates that either $\mathcal{C}$ has no solution or a solution for $\mathcal{C}$ could not be found.

Simplification and closure

This phase performs a trivial elimination of reflexive subtyping constraints. They are considered meaningless since they do not add information about types

restricted by them. Thus, these constraints are removed from the subtyping constraint set $\mathcal{C}$:

$$\{\tau <:_t \tau\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \mathcal{C}', err, \sigma$$

where $\tau \in \mathcal{T}y \backslash \{wt\}$.

The next property of the partial order $<:_t$ to be considered is the antisymmetric one. In order to solve a highest number of constraints by unification, we try to rewrite subtyping constraints into equality constraints as much as possible. For this purpose, we consider the following simplification rule:

$$\{\tau_1 <:_t \tau_2, \tau_2 <:_t \tau_1\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \mathcal{C}', \mathsf{solveEqConstraints}(\{\tau_1 =_t \tau_2\}, err, \sigma)$$

where $\tau_1, \tau_2, \tau_3 \in \mathcal{T}y \backslash \{wt\}$. We recall that the solveEqConstraints algorithm returns the generated set *err* and solution σ.

Type variables occurring in a subtyping constraint set can be indirectly limited due to the transitive property of $<:_t$. Thus, consequent constraints are computed by the application of transitive closure of $<:_t$ which also keeps the existing constraints of $\mathcal{C}$:

$$\{\tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \{\tau_1 <:_t \tau_2, \tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \cup \mathcal{C}', err, \sigma$$

where $\tau_1, \tau_2 \in \mathcal{T}y \backslash \{wt\}$ and $\alpha \in \mathcal{X}$.

The transitive closure of $<:_t$ on $\mathcal{C}$ represents the *closed form* of the subtyping constraint set that is stable via transitivity.

Definition 4.15 (Closed form)**.** *Let $\mathcal{C}$ be a subtyping constraint set. $\mathcal{C}$ is said to be* closed under transitivity*, or* closed *for short, if and only if whenever $\{\tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \uplus \mathcal{C}$ such that $\tau_1, \tau_2 \in \mathcal{T}y \backslash \{wt\}$ and $\alpha \in \mathcal{X}$, $\{\tau_1 <:_t \tau_2\}$ is defined and included in $\mathcal{C}$.*

Incompatibility detection

Since the hierarchy between types of terms is provided by the transitive closure of $<:_t$ on a context Γ, we define a rule to validate ground subtyping constraints. The objective is to detect which ground constraint of $\mathcal{C}$ are not valid with respect to the type hierarchy considered and, consequently, simplify the constraint set and possibly raise error messages.

$$\{s_1^{h_1} <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \mathcal{C}', \mathsf{isSub}(s_1^{h_1}, s_2^{h_2}) \cup err, \sigma$$

where $h_1, h_2 \in \mathcal{F}^\star \cup \{?\}$.

In this phase, the algorithm isSub, made of the rules presented in Figure 4.7, is called each time a ground subtyping constraint is found in $\mathcal{C}$. The rules for fail detection are based on the definition of the partial order $<:_t$ (see Definition 4.1) considering type hierarchy provided by Γ^*.

$$
\begin{array}{lllll}
\text{(1a)} & s_1^?, s_2^? & \Longrightarrow & \{\} & \text{if } (s_1^? <:_t s_2^?) \in \Gamma^* \\
\text{(1b)} & s_1^?, s_2^? & \Longrightarrow & \{(s_1^?, s_2^?)\} & \text{if } (s_1^? <:_t s_2^?) \notin \Gamma^* \\
\\
\text{(2)} & s_1^?, s_2^v & \Longrightarrow & \{(s_1^?, s_2^v)\} & \\
\\
\text{(3a)} & s_1^v, s_2^? & \Longrightarrow & \{\} & \text{if } (s_1^? <:_t s_2^?) \in \Gamma^* \\
\text{(3b)} & s_1^v, s_2^? & \Longrightarrow & \{(s_1^v, s_2^?)\} & \text{if } (s_1^? <:_t s_2^?) \notin \Gamma^* \\
\\
\text{(4a)} & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{\} & \text{if } (s_1^? <:_t s_2^?) \in \Gamma^* \wedge v_1 = v_2 \\
\text{(4b)} & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{(s_1^{v_1}, s_2^{v_2})\} & \text{if } (s_1^? <:_t s_2^?) \notin \Gamma^* \vee v_1 \neq v_2
\end{array}
$$

where $v, v_1, v_2 \in \mathcal{F}^\star$

Figure 4.7: The rules of algorithm isSub for fail detection in a subtyping constraint set $\mathcal{C}$.

Canonization

This phase aims to reduce the search space of subtyping constraint resolution. The idea is to put the subtyping constraint set in *canonical form*, i.e. to limit each type variable occurring in it with at most one ground type per bound.

Definition 4.16 (Canonical form of subtyping constraint set)**.** *Let $\mathcal{C}$ be a subtyping constraint set. $\mathcal{C}$ is said to be in* canonical form *if and only if each type variable α has at most one ground type as lower bound and one ground type as upper bound.*

In order to put an arbitrary subtyping constraint set in canonical form, we consider the following rules:

$$
\begin{array}{lcl}
\{s_1^{h_1} <:_t \alpha, s_2^{h_2} <:_t \alpha\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \{s^? <:_t \alpha\} \cup \mathcal{C}', err, \sigma \\
 & & \text{if } \exists s^? \text{ such that } s^? \text{ is the lub of } \{s_1^{h_1}, s_2^{h_2}\} \\
\{s_1^{h_1} <:_t \alpha, s_2^{h_2} <:_t \alpha\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \mathcal{C}', \{(s_1^{h_1}, s_2^{h_2})\} \cup err, \sigma \\
 & & \text{if } \nexists s^? \text{ such that } s^? \text{ is the lub of } \{s_1^{h_1}, s_2^{h_2}\} \\
\\
\{\alpha <:_t s_1^{h_1}, \alpha <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \{\alpha <:_t s_1^{h_1}\} \cup \mathcal{C}', err, \sigma \\
 & & \text{if } \Gamma^*(s_1^? <:_t s_2^?) \wedge (h_1 = h_2 \vee h_2 = ?) \\
\{\alpha <:_t s_1^{h_1}, \alpha <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \{\alpha <:_t s_2^{h_2}\} \cup \mathcal{C}', err, \sigma \\
 & & \text{if } \Gamma^*(s_2^? <:_t s_1^?) \wedge (h_1 = h_2 \vee h_1 = ?) \\
\{\alpha <:_t s_1^{h_1}, \alpha <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \mathcal{C}', \{s_1^{h_1}, s_2^{h_2}\} \cup err, \sigma \\
 & & \text{if } s_1^{h_1} \text{ and } s_2^{h_2} \text{ are not comparable}
\end{array}
$$

where $h_1, h_2 \in \mathcal{F}^\star \cup \{?\}$ and $\alpha \in \mathcal{X}$.

Errors caused by incompatible types also can be raised during the canonization process. This happens when two lower (resp. upper) bounds are not related and form a new pair (type,type) to be added to *err*.

Generation of solution

The subtyping constraint set $\mathcal{C}$ resulting from the sequence of simplification, closure, incompatibility detection and canonization phases (i.e. the propagation process) is said to be in *solved form* if no errors were found, that is $err = \{\}$. Thereafter we are interested in try to find a *solution* for $\mathcal{C}$.

Definition 4.17 (Solution for constraint set)**.** *Let $\mathcal{C}$ be a constraint set composed of subtyping constraints $\tau_1 <:_t \tau_2$. A* solution *of a subtyping constraint $\tau_1 <:_t \tau_2$ is a substitution σ such that $\sigma\tau_1 <:_t \sigma\tau_2$. A substitution σ is solution of $\mathcal{C}$ if it is solution of every subtyping constraint in $\mathcal{C}$.*

While solving $\mathcal{C}$ we wish to make sure, after each application of a rule for the generation of a solution, that the constraint set at hand is satisfiable, so as to detect errors as soon as possible. Therefore we combine the rules for constraint propagation with the following ones for constraint resolution:

$$\{\alpha <:_t s^h\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad [\alpha \mapsto s^h]\mathcal{C}', err, \{\alpha \mapsto s^h\} \circ \sigma$$

$$\{s^h <:_t \alpha\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad [\alpha \mapsto s^h]\mathcal{C}', err, \{\alpha \mapsto s^h\} \circ \sigma$$

where $h \in \mathcal{F}^\star \cup \{?\}$ and $\alpha \in \mathcal{X}$.

Instead of applying these rules exhaustively, only the first applicable one is applied. Thus, the order of the rules is really important as a means to opt for the least restrictive type when having two possibilities. For instance, in case a type variable α is limited by $\alpha <:_t s_1^{h_1}$ and $s_2^{h_2} <:_t \alpha$ in $\mathcal{C}$, the type $s_1^{h_1}$ will be assigned to α. As a type system for an embedded language, this approach aims to be compatible with the type system of Java in where upcasting is implicit and safe. This reduces conflicts with the type checking performed by Java that checks every cast done in the code generated by Tom.

Garbage collection

Many type variables are created during the generation of type constraints by the type inference algorithm. However, some of these type variables do not represent types of terms, but are used to indirectly limit types of input terms. In order to distinguish the two kinds of type variables, we define a set $\mathcal{I}$ of *input types*.

Definition 4.18 (Input types of a typing judgment)**.** *Consider a typing judgment $\varphi = \Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$. The set of* input types *of φ, denoted by $\mathcal{I}$, is a set that satisfies the following properties:*

- *if $e = \{rule_1; \ldots; rule_n\}$, then $\forall i \in [1,n], \forall \alpha \in \mathcal{I}, \exists \mathsf{x} \in \mathcal{V}, (\ \mathsf{x} : \alpha \in \Gamma_i')$ where $\Gamma = \Gamma' \bigcup_{i=1}^{n} \Gamma_i'$*
- *if $e \neq \{rule_1; \ldots; rule_n\}$, then $\mathcal{I} = \{\}$*

Although being useful for the transitive closure computation of $\mathcal{C}$, constraints limiting only non-input type variables that do not map input type variables are no longer necessary once the propagation phase is over. Thus, they are eliminated by the following rules:

$$\{\alpha_1 <:_t \alpha_2\} \uplus \mathcal{C}', err, \sigma, \mathcal{I} \implies \mathcal{C}', err, \sigma, \mathcal{I} \quad \text{if } \alpha_1, \alpha_2 \notin \sigma\mathcal{I}$$

$$\{\alpha_1 <:_t \alpha_2\} \uplus \mathcal{C}', err, \sigma, \mathcal{I} \implies \mathcal{C}', \{(\alpha_1, \alpha_2)\} \cup err, \sigma, \mathcal{I} \quad \text{if } \alpha_1 \in \sigma\mathcal{I} \vee \alpha_2 \in \sigma\mathcal{I}$$

where $\alpha_1, \alpha_2 \in \mathcal{X}$.

The garbage collecting works as a verification that all input type variables are in the domain of the solution σ. This means that all variables occurring in both numeric conditions and patterns of matching conditions have their types inferred. These expressions correspond to *pure* algebraic terms and the variables occurring in it are consequently *pure* Tom variables in contrast to variables that were possibly declared in Java code.

Here comes the definition of the subtyping constraint propagation and resolution algorithm solveSubConstraints.

Definition 4.19 (Subtyping constraint resolution algorithm)**.** *The algorithm* solveSubConstraints *takes as input a 4-tuple* $(\mathcal{C}, err, \sigma, \mathcal{I})$*, where* $\mathcal{C}$ *is a subtyping constraint set,* err *is a set of type errors initially empty,* σ *is a substitution initially empty and* $\mathcal{I}$ *is a set of type variables.*

The algorithm recursively applies the rules of simplification and closure phase until a fixpoint is reached. Then it loops over $\mathcal{C}$ *until* $\mathcal{C}$ *is not modified anymore. The incompatible detection phase is then initialized in the beginning of this loop. If the resulting* err *is non-empty, then the algorithm stops and raises out all type errors found. Otherwise, the canonization phase is started.*

Afterwards, the algorithm tries to apply one of the rules of generation of solution. If at least one of them is applicable, then the first applicable one is applied to $\mathcal{C}$ *and it goes back to the beginning of the loop. Otherwise, the algorithm interrupts the loop.*

Once out of the loop, the garbage collecting is activated until $\mathcal{C}$ *becomes empty. At this moment, if* err *is empty, then the generated* σ *constitutes a solution for the original* $\mathcal{C}$*. Otherwise, all error messages collected during the resolution have been raised and the algorithm fails because either the original* $\mathcal{C}$ *has no solution or more type annotations must be provided in order to find a solution for the original* $\mathcal{C}$*.*

We note that the application of one of the generation rules keeps $\mathcal{C}$ in closed form and does not generate neither reflexive constraints nor pairs of symmetric constraints. For that reason, the simplification and closure phase are not included in the loop.

Since type inference generates both an equality constraint set and a subtyping constraint set, constraint resolution is implemented by the algorithm solveConstraints which is made up of a combination of solveEqConstraints and solveSubConstraints. Initially, the equality constraint set and the empty sets err and σ are

given as arguments to solveEqConstraints. Then the returned sets err and σ in saturated form are verified. If $err \neq \emptyset$, then all errors messages collected during the equality constraint resolution have been raised and the algorithm solveConstraints fails since σ is rejected. Otherwise, σ is applied to the subtyping constraint set. The set $\mathcal{C}'$ obtained from the application of σ is, jointly with err, σ and a set $\mathcal{I}$ of input type variables, given to solveSubConstraints. The algorithm stops when the constraint set becomes empty. If $err \neq \emptyset$, then the algorithm fails and σ is rejected. Otherwise, σ is applied to the Tom expression considered in the constraint typing judgment.

Example 4.20. *Considering Example 4.11, in the constraint typing judgment*

$$\Gamma \vdash_{ct} \{\text{concNat(x*,pnat@!zero(),z*)} \ll |NatList^{?}|\ \text{nList} \longrightarrow \text{`pnat}\} : wt \bullet \mathcal{C}$$

the generated constraint set is $\mathcal{C} = \mathcal{C}_e \cup \mathcal{C}_s$ *where the equality constraint set is in canonical form*

$\mathcal{C}_e = \{\alpha_5 =_t NatList^{concNat}, \alpha_2 =_t \alpha_5, \alpha_4 =_t \alpha_7, \alpha_7 =_t ZNat^{?}, \alpha_3 =_t \alpha_5, \alpha_1 =_t \alpha_6, \alpha_4 =_t \alpha_8\}$

and the subtyping constraint set is $\mathcal{C}_s = \{\alpha_5 <:_t \alpha_6, \alpha_7 <:_t Nat^{?}\}$.

The generated set of input types is $\mathcal{C} = \{\alpha_2, \alpha_3, \alpha_4\}$.

For $\mathcal{C}_e$**:** *let* $err = \{\}$ *and* $\sigma = \{\}$. *Applying* solveEqConstraints *to* $(\mathcal{C}_e, err, \sigma)$, *we obtain* $err = \{\}$ *and*

$$\sigma = \{\alpha_5 \mapsto NatList^{concNat}, \alpha_2 \mapsto NatList^{concNat}, \alpha_4 \mapsto ZNat^{?}, \alpha_7 \mapsto ZNat^{?}, \alpha_3 \mapsto NatList^{concNat}, \alpha_1 \mapsto \alpha_6, \alpha_8 \mapsto ZNat^{?}\}$$

For $\mathcal{C}_s$**:** *let* $\mathcal{C}'_s = \sigma\mathcal{C}_s = \{NatList^{concNat} <:_t \alpha_6, ZNat^{?} <:_t Nat^{?}\}$. *The application of* solveSubConstraints *to* $(\mathcal{C}'_s, err, \sigma, \mathcal{I})$ *proceeds as in the following:*

- *Simplification and closure phase returns:*
$$(\{NatList^{concNat} <:_t \alpha_6, ZNat^{?} <:_t Nat^{?}\}, err, \sigma, \mathcal{I})$$
- *Incompatibility detection phase returns:*
$$(\{NatList^{concNat} <:_t \alpha_6\}, err, \sigma, \mathcal{I})$$
- *Canonization phase returns:*
$$(\{NatList^{concNat} <:_t \alpha_6\}, err, \sigma, \mathcal{I})$$
- *Generation of solution phase returns:*
$$(\{\}, err, \{\alpha_6 \mapsto NatList^{concNat}\} \circ \sigma, \mathcal{I})$$
- *Garbage collection phase returns:*
$$(\{\}, err, \{\alpha_6 \mapsto NatList^{concNat}\} \circ \sigma, \mathcal{I})$$

Since the constraint set $\mathcal{C}'_s$ *is empty, the algorihtm* solveConstraints *stops. Moreover,* $\{\alpha_1 \mapsto NatList^{?}, \alpha_2 \mapsto NatList^{concNat}, \alpha_3 \mapsto NatList^{concNat}, \alpha_4 \mapsto ZNat^{?}, \alpha_5 \mapsto NatList^{concNat}, \alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto ZNat^{?}, \alpha_8 \mapsto ZNat^{?}\}$ *is generated as a solution for* $(\Gamma, \{\text{concNat(x*,pnat@!zero(),z*)} \ll |NatList^{?}|\ \text{nList} \longrightarrow \text{`pnat}\}, wt, \mathcal{C})$, *since* $err = \emptyset$.

The algorithm solveConstraints always terminates, failing when given either a non-satisfiable constraint set as input or insufficient type annotations and otherwise returning a solution for the initial constraint set.

Theorem 4.21 (Termination of constraint resolution)**.** solveConstraints *halts, either failing or by returning a solution for the initial constraint set* $\mathcal{C}$*.*

Proof. Since solveConstraints corresponds to the application of solveEqConstraints and subsequent application of solveSubConstraints, we split this proof in two parts:

1. termination of solveEqConstraints,
2. termination of solveSubConstraints.

For part [1], the proof was already done in Theorem 3.37.

For part [2], we start by showing that the application of the rules of the simplification and closure phase terminate. The rules for simplification of reflexive and antisymmetric constraints decrease necessarily the number of type constraints of $\mathcal{C}$ until no constraints of the form $\tau <:_t \tau$ and no couples of the form $(\tau_1 <:_t \tau_2, \tau_2 <:_t \tau_1)$ can be found.

To show the termination of the application of the transitive rule, we first define some notations:

- $\mathcal{P}$ is the set of couples of subtyping constrains of $\mathcal{C}$ on which the transitive rule has been applied,
- $n_{\mathcal{P}}$ is the number of elements $\mathcal{P}$,
- n_α is the number of unique type variables occurring in $\mathcal{C}$,
- n_g is the number of unique ground types occurring in $\mathcal{C}$.

We denote by trans the transitive rule

$$\{\tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \uplus \mathcal{C}', err, \sigma \Longrightarrow \{\tau_1 <:_t \tau_2, \tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \cup \mathcal{C}', err, \sigma$$

A valid couple of arguments for the application of trans is a couple of the form $(\tau_1 <:_t \alpha, \alpha <:_t \tau_2)$. It can be seen as a 3-tuple (τ_1, α, τ_2). The set $\mathcal{P}$ is defined here to clarify the behavior of the recursive application of trans until a fix-point has been reached. $\mathcal{P}$ is initially empty and progressively incremented for each application of trans.

Let $n_{\mathcal{C}}$ be the upper bound of possible 3-tuples built from the types occurring in $\mathcal{C}$ to the application of trans. Since the application of trans does not affect the value of neither n_α nor n_g, then:

$$n_{\mathcal{C}} = (n_\alpha * n_g) * n_\alpha * (n_\alpha * n_g)$$

We note that any new constraint generated by the application of trans is already taken in account as a possible element of a 3-tuple argument.

We then proceed defining an interpretation $|\,.\,| : \mathcal{C} \times err \times \sigma \times \mathcal{P} \longrightarrow \mathbb{N}$ as $|\mathcal{C}, err, \sigma, \mathcal{P}| = n_{\mathcal{C}} - n_{\mathcal{P}}$. Thus, in the application of trans we have $|lhs(\text{trans})| = n_{\mathcal{C}} - n_{\mathcal{P}}$ and $|rhs(\text{trans})| = n_{\mathcal{C}} - (n_{\mathcal{P}} + 1)$. Hence, $|lhs(\text{trans})| > |rhs(\text{trans})|$ for each application of trans. Indeed, this shows that there is no infinite rewrite sequence of trans, that is, the application of the rules of the simplification and closure phase terminates.

The application of the rules of the remaining phases decrease necessarily the number of constraints of $\mathcal{C}$ until $\mathcal{C}$ becomes empty.

Thus, solveConstraints terminates. □

4.4 Implementation and integration

As previously presented in Section 3.4, the Tom system is divided into nine modules where the first one is the *parser*. Although being not related to the *typer* module responsible for type inference, the implementation of the new type system with subtyping started with the modification of the Tom syntax and, consequently, of the parsing phase. We added a new construct **extends** used to declared subtyping relations between algebraic types (i.e. base sorts) introduced by the **%typeterm** construct. Besides propagating the information about sort hierarchy in the sort table, the parser verifies the absence of multiple inheritance declarations.

The next step was the extension of the typer as a subtyping-constraint-based type system. Before performing type inference in blocks of a Tom program, each ground type (i.e. sorts decorated with ?) in the sort table is paired with a list containing all its distinct supertypes. The supertypes are calculated by the transitive closure of $<:_t$ and stored on a list named UpperTypeList. The resulting pairs (Type,UpperTypeList) are added to a hash map named Dependencies corresponding to Γ^*. This is done through the following pseudocode:

```
SET UpperTypeList to {}
FOR each Type in SortTable
    IF Type has a UpperType THEN
        SET UpperTypeList to {UpperType} U Dependencies[UpperType]
    ENDIF

    FOR each SubType in range of Dependencies
        IF Type is in Dependencies[SubType] THEN
            Overwrite pair (SubType, UpperTypeList U Dependencies[Subtype]) in Dependencies
        ENDIF
    ENDLOOP
    Add pair (Type,SubTypeList) to Dependencies
ENDLOOP
```

The typing process is implemented by the TopDownStopOnSucces strategy that controls the application of the type inference rules presented in Figures 4.4 and 4.5. The first rule to be applied is the [CT-Block] rule which collects the type variables assigned to variables occurring in numeric conditions as well as in patterns of matching conditions. These type variables are stored on a list $\mathcal{I}$ useful for the constraint resolution. Moreover, the constraints collected during type inference are grouped into two different lists $\mathcal{C}_e$ and $\mathcal{C}_s$ of equality and subtyping constraints, respectively. As happens with $\mathcal{C}_e$, the insertion of a constraint c in $\mathcal{C}_s$ is done only if c is not in $\mathcal{C}_s$. When the end of a block is reached, the algorithm solveEqConstraints is called to loop over $\mathcal{C}_e$ and generates a substitution σ in saturated form and a flag variable err. In case err has the value false at the end of $\mathcal{C}_e$, the resulting substitution is applied to $\mathcal{C}_s$ generating $\mathcal{C}'_s$ such that $\mathcal{C}'_s = \sigma\mathcal{C}_s$.

Constraint resolution proceeds on $\mathcal{C}'_s$ by performing constraint propagation, generation of solution and garbage collecting. The rules considered in the sim-

plification and closure phase have their application controlled by the RepeatId strategy:

$$(\mathcal{R}epeatid\ \mathsf{simplifyAndClose})(e)$$
$$= (\mathcal{S}eqid\ \mathsf{simplifyAndClose}\ (\mathcal{R}epeatid\ \mathsf{simplifyAndClose}))(e)$$

where the strategy $\mathcal{S}eqid$ is similar to $\mathcal{S}eq$, presented in Section 2.2.4, except it returns e instead of throwing an exception in case there exists no expression e' such that $e \neq e'$ and $\mathcal{S}eq(e)$ reduces to e'.

After the transitive closure of $\mathcal{C}'_s$, the rule of incompatibility detection is folded over the resulting constraint set. As happens with isEq, the implementation of the auxiliary algorithm isSub is slightly different from the description of Figure 4.7, since *err* is not a set of pairs (type,type) but a flag (boolean) variable. In case a type error is detected, the algorithm outputs an error message and sets *err* to true. Otherwise, *err* is assigned to false. At the end of $\mathcal{C}'_s$, the value of *err* determines whether the canonization phase is started. In the negative case, the constraint resolution is interrupted without generating a valid solution. On the contrary, the rules for canonization are applied under the RepeatId strategy.

In order to apply the first applicable rule of generation of solution on $\mathcal{C}'_s$, a list-matching is performed on $\mathcal{C}'_s$ searching initially for a constraint $\alpha <:_t s^h$. If it holds, then a break statement interrupts this phase. Otherwise, it searches for a constraint $s^h <:_t \alpha$ until to reach the end of $\mathcal{C}'_s$. List-matching is also used to implement the garbage collection phase ensuring that all constraints involving only non-input type variables are eliminated from $\mathcal{C}'_s$.

Altogether the algorithm solveSubConstraints as described in Definition 4.19 is called to order the phases of constraint resolution in $\mathcal{C}'_s$. In practice, it takes only a list named ConstraintList with all constraints of $\mathcal{C}'_s$ as input since σ and $\mathcal{I}$ are global variables and *err* is a flag variable declared and used in the incompatibility detection phase, as previously mentioned. The pseudocode of the algorithm is the following:

```
GET ConstraintList
SET ConstraintList to the application of the simplification and closure in it
REPEAT
    SET ConstraintList to the appplication of incompatibility detection in it
    IF Err is true THEN
        RETURN ConstraintList
    ELSE
        SET ConstraintList to the application of canonization in it
        SET ConstraintList to the application of generation of solution in it
    ENDIF
UNTIL ConstraintList has reached a fixpoint
CALL garbage collection in ConstraintList
```

After solving equality and subtyping constraints, if the value of *err* is **true** then σ is rejected. Otherwise, σ is applied to the AST node for which the type inference was performed. This process is relaunched until the end of the AST

input, when the typer replaces possible remaining type variables by an internal type EmptyType stating that it could not be inferred.

The *compiler* module was also modified in order to introduce a type verification for variable patterns conforming to the rules of Figure 4.1. As the transformations performed in compilation, the type verification is expressed in an intermediate language. Then, the *backend* module translates it in dynamic tests that use the Java runtime operator instanceof to check casts done in the Java generated code. A performance analysis of the Tom system obtained from the addition of subtyping is provided in the next chapter.

4.5 Synthesis

In order to enrich the type system of the Tom language, we proposed the introduction of *subtyping* as a partial order over types representing Tom algebraic data types. Consequently, the notion of type compatibility was weakened. As a nominal type system, the possibility to declare a type hierarchy had an impact on the evaluation of matching conditions since **%match** constructs involve explicit type annotations to be considered. This was considered in the semantic rules describing acceptance and rejection of matching conditions whose patterns are variables.

The presence of subtyping declarations allows terms to be typed with subtypes of those indicated in their type annotations. For this purpose we introduced a subsumption rule to be combined with the remaining rules of type checking. Furthermore, we spread subtyping constraints on the rules of the constraint-based type inference system, particularly on rules handling matching and numeric conditions as well as on those for ground terms. This led to the generation of a constraint set which is split into two other ones: an equality and a subtyping constraint set. The former set is solved by rules of unification. For the resolution of subtyping constraints, we defined rules for a combination of constraint propagation, generation of solution and garbage collecting of remaining constraints.

The resulting subtyping-constraint-based type system proposed in this chapter is characterized by the support of associative pattern matching on algebraic terms representing objects. Its particularity lies on its ability to manipulate algebraic terms built from variadic operators. The type inference on the arguments of these terms requires a notion of types including enough information to the distinction of different variadic operators. Moreover, in the context of term rewriting embedded in object-oriented programming, our type system deals with type inclusion at the pattern level and is able to infer types compatible with the ones expected by the type system of the host language. It was shown to be safe and subsequently implemented and integrated in the current Tom version 2.9 available at the Tom source repository[8]. In addition to the test cases, the resulting Tom compiler was successfully bootstrapped using the typer. To

8. Reference available at https://gforge.inria.fr/projects/tom/ . Visited on February 2012.

demonstrate the efficiency of our type system, in Chapter 5 we present the results of comparative experiments between the current and the previous Tom compiler versions together with its use in the transformation of EMF models.

Chapter 5

Typing in practice

The main advantages due to the integration of the subtyping-constraint-based type system formally defined in Chapter 4 are discussed in the present chapter. For this purpose, we present some simple examples written in Tom and use more complex examples to benchmark the performance of the resulting typer. Although leading to a slower typer (and consequently compiler), it provides more type safety to the Tom system. In order to illustrate the application of type inclusion in Tom mappings and consequently at pattern level, we simultaneously present both a hierarchical and a non-hierarchical (i.e. with a single level of type hierarchic) mappings implemented by Java classes. Besides, a real application of inheritance structures to pattern matching is described by a case study of model transformation using Tom.

5.1 Motivation

As explained in Section 2.2, the Tom language provides pattern matching and rule based constructs to Java through the use of the **%match** construct. This construct is applied over algebraic terms independently of their concrete representation in Java (owing to the definition of mappings). Mappings are used by the ‘ (backquote) construct to build algebraic terms that can be manipulated in the scope of Java code. However, although both **%match** and ‘ are constructs of the Tom core syntax (see Definition 3.5), they were not homogeneously considered by the Tom type system available in the beginning of this thesis. Therewith the typer was able to infer types of variables occurring in the patterns of conditions of rewrite rules and propagate them to the occurrences (if any) of these variables in backquote constructs of the action. Accordingly, considering the Gom algebraic signature defined in module SimpleGom

```
1  %gom {
2    module SimpleGom
3    abstract syntax

5    Numeral = zero()
```

```
6             | suc(n:Numeral)
7             | sum(Numeral*)
8             | product(Numeral*)

10    Letter = vowel()
11            | syllable (a1:Letter,a2:Letter)
12  }
```

the types of the occurrences of x at lines 15 and 16 of the following pattern matching construct could not be inferred since both patterns and subjects are variables and the type parameter of **%match** is not explicitly declared:

```
13  public void simpleMatch(Numeral num1, Numeral num2) {
14    %match {
15      x << num1                          -> { System.out.println(`suc(num1)); }
16      x << num1                          -> { System.out.println(`x); }
17      product(x*,y) << num2              -> { System.out.println(`x); }
18      sum(product(zero()),x) << num2     -> { System.out.println(`x); }
19      sum(product(zero()),x*) << num2    -> { System.out.println(`x); }
20    }
21  }
```

The type system presented in Section 4.3 overcomes the necessity of providing type annotations for whichever variables occurring in matching/numeric conditions. It considers algebraic terms in conditions and actions with the same level of importance. The type of variable x in line 15 is inferred by the type information of the domain of suc in right-hand side. Moreover, the type of the subject num1 is inferred by the type of the pattern x. In line 16, the method println occurring in the action side is not useful to infer the type of x since it can take arguments of whichever type. In this case, the type of x is inferred by the type of variable num1 occurring in the subject which is the same of the one at line 15. In line 17, the domain of product is used to infer types of x, y and num2. Lines 18 and 19 look similar but while in the latter case x can be instantiated only by a list with head symbol sum, the former considers either a list or a function operator with the codomain sort Number as valid instances.

Besides bringing more power to the Tom system, the constraint-based type inference system with/without subtyping is also able to detect type incompatibilities for a larger number of algebraic terms. For instance, the following Tom code which was previously considered as well-typed presently gives rise to type errors for each one of the matching conditions:

```
22  public void checkTypes(Numeral num1, Numeral num2) {
23    %match {
24      sum(x*) << num1 && product(x*) << num2 -> {
25        System.out.println("Incompatible types 'Numeral' and 'Numeral' for 'x'.");
26      }

28       syllable (x,y) << num1 -> {
29        System.out.println("Out of the inner match 'x' has type 'Letter'");
30        %match {
```

```
31            product(x) << num2 -> {
32              System.out.println("Incompatible types 'Numeral' and 'Letter' for 'x'.");
33            }
34          }
35        }
36      }
37    }
```

The method checkTypes contains two type errors which were previously not detected by the Tom compiler but by the Java compiler. The first type error is found on line 24 arises because the non-linear star variable x* is used as argument of two lists with different head symbols, that is incompatible types $\mathsf{Numeral}^{sum}$ and $\mathsf{Numeral}^{product}$ are inferred for x. Likewise, for the second type error, incompatible types are inferred for the non-linear variable x occurring in the embedded %match construct of line 30.

In addition to safety, the subtyping-constraint-based type system provides more expressiveness to the Tom language since it allows type hierarchies to be considered when performing matching in Java. Although the type system with subtyping has been formally defined in previous chapters, examples and benchmarks may allow a better understanding of the improvements obtained by the Tom language and system.

5.2 Example

Algebraic data types can be organized in a type hierarchy through subtyping declarations. In order to draw a comparison between programming with and without subtyping in Tom, we present a classical example of algebraic data types: the abstract syntax of symbolic expressions and statements. As usual, we define expressions and statements as distinct types and their inheritance structures as in Figure 5.1.

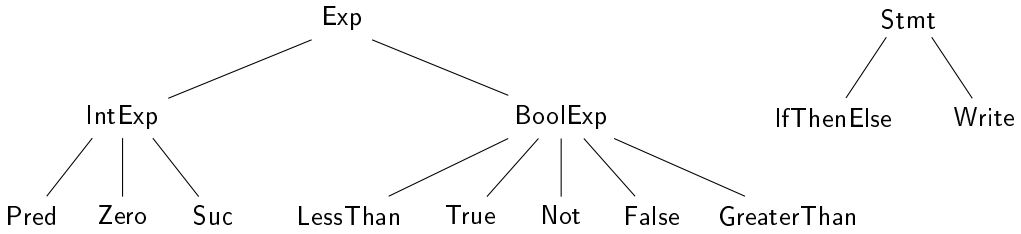


Figure 5.1: A type hierarchy for symbolic expressions and statements.

We describe an example of a symbolic expression evaluator and statement executor encoded by two extensionally equivalent Java classes. They include handwritten mappings for algebraic data types and perform pattern matching over algebraic terms. Since mappings are composed of declarations of the algebraic data types and their respective implementations, we first describe the

Java classes of concrete types and operators for expressions and statements. The concrete types are organized in an inheritance structure on the left column and in a flat structure on the right column where classes IntExp and BoolExp were discarded:

```
public static abstract class Stmt {}
public static abstract class Exp {}
public static abstract class IntExp extends Exp {}
public static abstract class BoolExp extends Exp {}

static class IfThenElse extends Stmt {
  public BoolExp b;
  public Stmt s1;
  public Stmt s2;
  public IfThenElse(BoolExp b, Stmt s1, Stmt s2) {
    this.b = b; this.s1 = s1; this.s2 = s2;
  }
}

static class Write extends Stmt {
  public Exp e;
  public Write(Exp e) { this.e = e; }
}

static class Zero extends IntExp {
  public Zero() {}
}

static class Suc extends IntExp {
  public IntExp n;
  public Suc(IntExp n) { this.n = n; }
}

static class Pred extends IntExp {
  public IntExp n;
  public Pred(IntExp n) { this.n = n; }
}

static class True extends BoolExp {
  public True() {}
}

static class False extends BoolExp {
  public False() {}
}

static class Not extends BoolExp {
  public BoolExp b;
  public Not(BoolExp b) { this.b = b; }
}

static class LessThan extends BoolExp{
  public IntExp n1;
  public IntExp n2;
  public LessThan(IntExp n1, IntExp n2) {
    this.n1 = n1; this.n2 = n2;
  }
}

static class GreaterThan extends BoolExp {
  public IntExp n1;
  public IntExp n2;
  public GreaterThan(IntExp n1, IntExp n2) {
    this.n1 = n1; this.n2 = n2;
  }
}
```

```
public static abstract class Stmt {}
public static abstract class Exp {}



static class IfThenElse extends Stmt {
  public Exp b;
  public Stmt s1;
  public Stmt s2;
  public IfThenElse(Exp b, Stmt s1, Stmt s2) {
    this.b = b; this.s1 = s1; this.s2 = s2;
  }
}

static class Write extends Stmt {
  public Exp e;
  public Write(Exp e) { this.e = e; }
}

static class Zero extends Exp {
  public Zero() {}
}

static class Suc extends Exp {
  public Exp n;
  public Suc(Exp n) { this.n = n; }
}

static class Pred extends Exp {
  public Exp n;
  public Pred(Exp n) { this.n = n; }
}

static class True extends Exp {
  public True() {}
}

static class False extends Exp {
  public False() {}
}

static class Not extends Exp {
  public Exp b;
  public Not(Exp b) { this.b = b; }
}

static class LessThan extends Exp {
  public Exp n1;
  public Exp n2;
  public LessThan(Exp n1, Exp n2) {
    this.n1 = n1; this.n2 = n2;
  }
}

static class GreaterThan extends Exp {
  public Exp n1;
  public Exp n2;
  public GreaterThan(Exp n1, Exp n2) {
    this.n1 = n1; this.n2 = n2;
  }
}
```

We consider a mapping with and without subtyping on the left and on the right columns, respectively. In the mapping with subtyping we declare an algebraic type Exp and its algebraic subtypes IntExp and BoolExp. However, as

happens with concrete types, only a single algebraic type Exp is declared in addition to Stmt in the mapping without subtyping:

```
%typeterm Stmt {
  implement  { Stmt }
  is_sort(t)  { t instanceof Stmt }
}

%typeterm Exp {
  implement  { Exp }
  is_sort(t)  { t instanceof Exp }
}

%typeterm IntExp extends Exp {
  implement  { IntExp }
  is_sort(t)  { t instanceof IntExp }
}

%typeterm BoolExp extends Exp {
  implement  { BoolExp }
  is_sort(t)  { t instanceof BoolExp }
}
```

```
%typeterm Stmt {
  implement  { Stmt }
  is_sort(t)  { t instanceof Stmt }
}

%typeterm Exp {
  implement  { Exp }
  is_sort(t)  { t instanceof Exp }
}
```

The remaining Java classes are mapped to algebraic operators in both mappings but with different types in each one of them. While the mapping with subtyping respects the type hierarchy of Figure 5.1, all operators are of type either Exp or Stmt in the opposite mapping (in the right column). Therefore, data constructors IfThenElse and Write are of type Stmt in both columns, but the first argument of the former data constructor is of type BoolExp in the left column and Exp in the right one:

```
%op Stmt IfThenElse(b:BoolExp,s1:Stmt, s2:Stmt) {
  is_fsym(t)      { t instanceof IfThenElse }
  get_slot(b,t)   { ((IfThenElse)t).b }
  get_slot(s1,t)  { ((IfThenElse)t).s1 }
  get_slot(s2,t)  { ((IfThenElse)t).s2 }
  make(t0,t1,t2)  { new IfThenElse(t0,t1,t2) }
}

%op Stmt Write(e:Exp) {
  is_fsym(t)     { t instanceof Write }
  get_slot(e,t)  { ((Write)t).e }
  make(t0)       { new Write(t0) }
}
```

```
%op Stmt IfThenElse(b:Exp,s1:Stmt, s2:Stmt) {
  is_fsym(t)      { t instanceof IfThenElse }
  get_slot(b,t)   { ((IfThenElse)t).b }
  get_slot(s1,t)  { ((IfThenElse)t).s1 }
  get_slot(s2,t)  { ((IfThenElse)t).s2 }
  make(t0,t1,t2)  { new IfThenElse(t0,t1,t2) }
}

%op Stmt Write(e:Exp) {
  is_fsym(t)     { t instanceof Write }
  get_slot(e,t)  { ((Write)t).e }
  make(t0)       { new Write(t0) }
}
```

The remaining data constructors are all of type Exp in the mapping without subtyping:

```
%op IntExp Zero() {
  is_fsym(t) { t instanceof Zero }
  make()     { new Zero() }
}

%op IntExp Suc(n:IntExp) {
  is_fsym(t)    { t instanceof Suc }
  get_slot(n,t) { ((Suc)t).n }
  make(t0)      { new Suc(t0) }
}

%op IntExp Pred(n:IntExp) {
  is_fsym(t)    { t instanceof Pred }
  get_slot(n,t) { ((Pred)t).n }
  make(t0)      { new Pred(t0) }
}

%op BoolExp True() {
  is_fsym(t) { t instanceof True }
  make()     { new True() }
}

%op BoolExp False() {
  is_fsym(t) { t instanceof False }
  make()     { new False() }
}

%op BoolExp Not(b:BoolExp) {
  is_fsym(t)    { t instanceof Not }
  get_slot(b,t) { ((Not)t).b}
  make(t0)      { new Not(t0) }
}

%op BoolExp LessThan(n1:IntExp, n2:IntExp) {
  is_fsym(t)     { t instanceof LessThan }
  get_slot(n1,t) { ((LessThan)t).n1 }
  get_slot(n2,t) { ((LessThan)t).n2 }
  make(t0,t1)    { new LessThan(t0,t1) }
}

%op BoolExp GreaterThan(n1:IntExp, n2:IntExp) {
  is_fsym(t)     { t instanceof GreaterThan }
  get_slot(n1,t) { ((GreaterThan)t).n1 }
  get_slot(n2,t) { ((GreaterThan)t).n2 }
  make(t0,t1)    { new GreaterThan(t0,t1) }
}
```

```
%op Exp Zero() {
  is_fsym(t) { t instanceof Zero }
  make()     { new Zero() }
}

%op Exp Suc(n:Exp) {
  is_fsym(t)    { t instanceof Suc }
  get_slot(n,t) { ((Suc)t).n }
  make(t0)      { new Suc(t0) }
}

%op Exp Pred(n:Exp) {
  is_fsym(t)    { t instanceof Pred }
  get_slot(n,t) { ((Pred)t).n }
  make(t0)      { new Pred(t0) }
}

%op Exp True() {
  is_fsym(t) { t instanceof True }
  make()     { new True() }
}

%op Exp False() {
  is_fsym(t) { t instanceof False }
  make()     { new False() }
}

%op Exp Not(b:Exp) {
  is_fsym(t)    { t instanceof Not }
  get_slot(b,t) { ((Not)t).b}
  make(t0)      { new Not(t0) }
}

%op Exp LessThan(n1:Exp, n2:Exp) {
  is_fsym(t)     { t instanceof LessThan }
  get_slot(n1,t) { ((LessThan)t).n1 }
  get_slot(n2,t) { ((LessThan)t).n2 }
  make(t0,t1)    { new LessThan(t0,t1) }
}

%op Exp GreaterThan(n1:Exp, n2:Exp) {
  is_fsym(t)     { t instanceof GreaterThan }
  get_slot(n1,t) { ((GreaterThan)t).n1 }
  get_slot(n2,t) { ((GreaterThan)t).n2 }
  make(t0,t1)    { new GreaterThan(t0,t1) }
}
```

Once the mappings are defined for algebraic types and operators, we can now write Java methods that perform pattern matching on algebraic terms. We initially define the compute method to execute statements:

```
public void compute(Stmt stmt) {
  %match {
    IfThenElse(b,m1,m2) << stmt −> {
      if (evalBool(‘b)){
        compute(‘m1);
      } else { compute(‘m2); }
    }

    Write(e) << stmt &&
      x << IntExp e −> {
        System.out. println ( evalInt (‘x));
      }

    Write(e) << stmt &&
      x << BoolExp e −> {
        System.out. println (evalBool (‘x));
      }
  }
}
```

```
public void compute(Stmt stmt) {
  %match {
    IfThenElse(b,m1,m2) << stmt &&
      (True|False|Not|LessThan|GreaterThan)[] << b −> {
        if (evalBool(‘b)){
          compute(‘m1);
        } else { compute(‘m2); }
      }

    Write(e) << stmt &&
      (Zero|Suc|Pred)[] << e −> {
        System.out. println ( evalInt (‘e));
      }

    Write(e) << stmt &&
      (True|False|Not|LessThan|GreaterThan)[] << e −> {
        System.out. println (evalBool (‘e));
      }
  }
}
```

Since IntExp and BoolExp appearing on the left column are in conformity with the previously defined Java type hierarchy, these type constructors are used as explicit type parameters of pattern matching. The execution of an statement depends on the evaluation of the expressions occurring in it. This is encoded by the methods evalInt and evalBool which are identical in both versions of the current example:

```
public int evalInt (Exp e) {
  %match(e) {
    Zero()          −> { return 0; }
    Suc(x)          −> { return evalInt(‘x) + 1; }
    Pred(x)         −> { return evalInt(‘x) − 1; }
  }
  throw new RuntimeException("Ill−formed int exp' " + e + "'.");
}

public boolean evalBool(Exp e) {
  %match(e) {
    True()          −> { return true; }
    False()         −> { return false; }
    Not(x)          −> { return evalBool(‘x); }
    LessThan(x,y)   −> { return evalInt(‘x) < evalInt(‘y); }
    GreaterThan(x,y) −> { return evalInt(‘x) > evalInt(‘y); }
  }
  throw new RuntimeException("Ill−formed boolean exp' " + e + "'.");
}
```

We note that the definition of a type hierarchy allows patterns to match over subtypes. This happens in last two pattern matching cases of compute method from the left column. Moreover, the use of subtypes provides a way to statically and automatically catch type errors which reduces errors at runtime. For instance, the following code is considered ill-typed in the subtyping version of the example:

```
int result = evalInt(`Suc(True()));
```

However, in the example with flat structured mapping, the term Suc(True()) is considered well-typed but raises an unexpected java exception during evaluation. This runtime exception is due to the fact that no pattern matching cases in the evalInt method handles the term.

5.3 Performance analysis

The implementation of the subtyping-constraint-based type system in Tom had a positive impact on legibility due to the documentation written in comments. Although this leads to many lines of code, code documentation can be generated using the Javadoc tool [9] where both typing and constraint resolution rules are explained for methods implementing them.

The type system with subtyping proposed in this thesis proved to be type safe (see Section 4.3.2) and more expressive than its previous versions since it allows subtypes to be declared. For that reason, the subtyping-constraint-based type system must become the default typer of the next 2.10 Tom release to be available soon. Besides allowing the representation of inheritance structures holding algebraic data types, the formalization of the type system delimited the tasks to be performed only by the typing phase of the Tom system and no other phase. It resulted in performing type inference for a larger number of Tom constructs such as subjects of pattern matching problems and backquoted terms. Consequently, type inference and total compilation time increased considerably for larger applications. Since the Tom system is quite large and is implemented in Tom by a bootstrap technique, we consider its self-compilation as an interesting benchmark:

	Typing	Compilation	#Constraints	#Fresh type variables
version 2.9	59s	5min 40s	0	0
version 2.9 -nt	7min 55s	12min 40s	8,461,954	195,692

A total of 86 Tom files are compiled during a bootstrap containing a sum of subjects and backquoted terms equal to 796,182. The version 2.9 corresponds to the current development Tom system where the option "-nt" activates the subtyping-constraint-based type inference. The evaluation was done on a Mac OS X Server 10.6.8 with Darwin kernel version 10.8.0 and offering two 2.26 GHz Quad Core Intel Xeon processors with 24 GB of RAM memory. The compiler versions used were Java 1.6.0_26 and gcc 4.2.1. We also benchmarked the performance of our type system against some of the examples available on the Tom source repository [10]:

9. Reference available at http://docs.oracle.com/javase/1.4.2/docs/tooldocs/javadoc/ . Visited on February 2012.

10. Reference available at https://gforge.inria.fr/projects/tom/ . Visited on February 2012.

			version 2.9		version 2.9 -nt					
	#F	#B/S	Cmp.	Typ.	Cmp.	Typ.	#Ct.	#Var	ΔCmp.	ΔTyp.
acmatching	2	87	0.40 s	0.03 s	0.45 s	0.06 s	83	30	12%	100%
addressbook	4	122	1.18 s	0.04 s	1.29 s	0.13 s	1,841	176	9%	225%
analysis	9	1,058	4.33 s	0.57 s	5.14 s	2.44 s	6,570	1,037	19%	328%
antipattern	3	326	1.54 s	0.20 s	2.63 s	1.27 s	1,677	370	71%	535%
associative	2	553	1.67 s	0.29 s	2.32 s	1.35 s	5,289	542	39%	366%
baralgo	15	7,104	2.63 s	0.74 s	4.17 s	2.94 s	52,924	6,106	59%	297%
bdd	2	432	1.25 s	0.23 s	1.97 s	0.99 s	2,340	319	58%	330%
boulderdash	1	42	0.33 s	0.02 s	0.35 s	0.04 s	74	32	6%	100%
builtin	13	667	1.71 s	0.08 s	1.65 s	0.13 s	4,945	278	-4%	62%
cell	2	1,377	1.09 s	0.21 s	17.54 s	14.67 s	83,763	2,345	1509%	6886%
checking	2	323	1.64 s	0.22 s	2.30 s	0.98 s	1,606	420	40%	345%
csmaca	1	619	1.55 s	0.55 s	2.17 s	0.43 s	8,890	782	40%	-22%
debugger	8	157	1.11 s	0.20 s	1.97 s	0.93 s	381	694	77%	365%
diffterms	2	1,581	0.29 s	0.09 s	0.46 s	0.30 s	10,612	1,340	59%	233%
disunification	1	879	1.88 s	0.46 s	2.00 s	0.50 s	14,149	1,898	6%	9%
gomoku	3	795	2.55 s	0.53 s	8.78 s	5.29 s	45,564	1,990	244%	898%
graph	1	448	0.18 s	0.04 s	0.46 s	0.27 s	1246	488	156%	575%
inference	2	1,146	2.50 s	0.53 s	3.04 s	1.31 s	7,242	1,350	22%	147%
lambdacalculi	10	381	3.04 s	0.33 s	4.98 s	1.35 s	1,691	610	64%	309%
lazyml	9	7,187	8.57 s	2.62 s	22.67 s	17.54 s	51,457	3,954	165%	569%
list	5	324	0.61 s	0.05 s	0.62 s	0.06 s	525	112	2%	20%
master	41	10,410	7.39 s	0.65 s	9.15 s	1.00 s	50,845	2,274	24%	54%
matching	1	59	1.01 s	0.01 s	0.70 s	0.04 s	175	8	-31%	300%
mgs	3	253	0.52 s	0.06 s	0.58 s	0.10 s	338	94	12%	67%
miniml	6	1,953	4.34 s	1.01 s	9.40 s	6.36 s	13,827	1,475	117%	530%
modeltrans	8	189	2.61 s	0.52 s	4.19 s	1.65 s	1,990	662	61%	217%
nodisunification	2	1,866	0.49 s	0.14 s	0.73 s	0.39 s	29,772	1,665	49%	179%
peano	5	234	0.45 s	0.04 s	0.45 s	0.05 s	587	79	0%	25%
poly	10	4,476	2.38 s	0.43 s	3.31 s	1.42 s	23,568	1,627	39%	230%
polygraphes	2	135	1.07 s	0.14 s	1.66 s	0.54 s	902	242	55%	286%
prodrule	5	750	1.96 s	0.18 s	2.13 s	0.13 s	2,842	310	9%	-28%
propp	3	684	2.00 s	0.35 s	2.42 s	0.88 s	3,825	587	21%	151%
quine	1	3	0.29 s	0.01 s	0.30 s	0.02 s	13	10	3%	100%
rbtree	2	307	1.45 s	0.12 s	1.48 s	0.25 s	3,004	300	2%	108%
set	1	257	1.14 s	0.16 s	1.45 s	0.42 s	940	277	27%	162%
strategy	24	4,655	3.37 s	0.61 s	17.86 s	11.97 s	16,062	4,862	430%	1862%
strings	1	9	0.29 s	0.01 s	0.30 s	0.03 s	35	22	3%	200%
structure	1	73	1.50 s	0.19 s	1.76 s	0.50 s	723	338	17%	163%
termgraph	12	927	3.27 s	0.72 s	7.58 s	4.71 s	9,581	1,250	132%	554%
tomjastadd	1	21	0.68 s	0.06 s	1.30 s	0.66 s	163	54	91%	1000%
typeinference	36	4,666	2.55 s	0.21 s	3.32 s	0.49 s	15,797	859	30%	133%

Each row of the table is related to an example directory that contains a number of files indicated in column labeled "#F". In addition to the total of backquoted terms and subjects given in column "#B/S", another criteria that proportionally affect typing and compilation time (respectively in "Typ." and "Cmp." columns) are the number of rules *cond* $\longrightarrow$ *action* in a **%match** block and their respective number of (matching and numeric) conditions. Examples cell, lazyml and strategy illustrate the performance decrease of typing time in the presence of large

%match blocks with composed conditions. It occurs because of the large amount of generated constraints (in "#Ct." column), mainly subtyping constraints, between type variables (in "#Var" column) in a single constraint set to be traversed by strategies of the constraint resolution algorithm (see Section 4.4). The delta values (ΔCmp. and ΔTyp.) indicate the percentage of the difference between compilation time as well as typing time of examples using Tom version 2.9 with and without activated subtyping-constraint-based type system.

5.4 Application to model transformation

In recent years, the *Model-Driven Engineering* (MDE) has become an indispensable tool to understand and overcome computer systems getting more and more complex. It comes from the fact that software modeling can be used to both develop and verify software through *models* and *metamodels*. A *model* is an abstraction of an object. It must be in conformity with its *metamodel* that is the representation of a particular viewpoint on data structures. Accordingly, we may conceive the adoption of the Tom language as a model to manipulate abstract data structures, i.e. metamodels. For this purpose, we exploit the modeling framework EMF [SBPM09] which consists in a code generation environment for building tools based on a data structure. It implements the *Object Management Group* (OMG) *Essential Meta-Object Facility* (EMOF) standard for MDE. The model used to represent models in EMF is called Ecore that is, itself, an EMF model and consequently its own metamodel. Thus, metamodels developed with EMF must be in conformity with the Ecore metamodel.

5.4.1 Tom-EMF

The combination of Tom and EMF has been elaborated under the quarteFt project [11]. Its main goal is to develop technologies to facilitate the development of critical systems in the context of real-time embedded systems. The development life cycle is based on Domain Specific Modeling Languages (DSML) and also on certified model transformations between languages with respect to some syntactic and semantic properties. The quarteFt project proposes the development of techniques of specification and implementation of transformations as well as formal methods to validate and check them. It requires then the extension of the Tom language with constructs to define transformations. Being an embedded language, Tom allows model transformations to be described directly in Java, differently from the other existing transformation languages as ATL [AI04] or Kermeta [JBF11, DFF+10].

The Tom-EMF combination provides automatic generation of mappings for Ecore metamodels. As abstract data structures, the manipulation of these metamodels is supported in Tom by three components:

11. Reference available at http://quarteft.loria.fr . Visited on February 2012.

- a generator of Tom mappings between the Java class hierarchies generated by EMF and the Tom algebraic data types representing metamodels,
- a generator of XMI files to enable the interchange of model information during the model transformation,
- a specialized introspector distributed as a Java library to enable the application of Tom strategies to those terms built from the generated Tom mappings.

The implementation of a type system with nominal subtyping in Tom had a direct and strong impact on the development of the mapping generator for Ecore metamodels since it deals with Java inheritance. Ecore metamodels are expressed in a subset of the Unified Modeling Language [RJB99] (UML) composed of constructs used in class diagrams. The straightforward mapping from UML to Ecore is defined by EMF whose code generator creates the Java implementation of the input metamodels. These models can be algebraically specified through the definition of Tom mappings implemented by the generated Java classes. For the sake of simplicity and ease of comprehension, instead of consider Ecore elements that map UML constructs, we explain how the Tom-EMF mapping generator specifies Tom algebraic data types and subtypes representing the elements of UML class diagrams. Interested readers are referred to Chapter 6 in the book of Steinberg et al. [SBPM09] for a detailed description of the mapping of UML to Ecore defined by EMF.

The generator of Tom mappings covers all constructs of UML class diagrams:

Packages Each UML package is mapped to a Tom mapping file with the same name,

Classes and interfaces A type and a data constructors are created to every concrete class and interface in the model through the use of **%typeterm** and **%op** constructs, respectively. Every type constructor which is not a subtype of another algebraic type extends the EObject universal superclass in Ecore. The class attributes are mapped to data constructors by using either **%op** or **%oplist** depending on whether they are respectively single-valued or multi-valued. Abstract classes are mapped to type constructors also extending EObject. However, their data constructors are those defined for their subtypes,

Enumerations Each enumeration is mapped to an algebraic type whose implementation class does not extend any other class. Enumeration attributes are in their turn mapped to data constructors,

Data types The UML classes with a «datatype» stereotype correspond to the built-in types of data in Java. Accordingly, their corresponding Gom module must be imported in the mapping as explained in Section 2.2.1,

Class relationships Inheritance relationships are represented in the Tom mappings by a declaration of subtype relation between algebraic types. Although UML and Ecore specification supports multiple inheritance, EMF transforms it in single inheritance because of Java limitations. There are also two forms of general relationships to be considered: non-containment

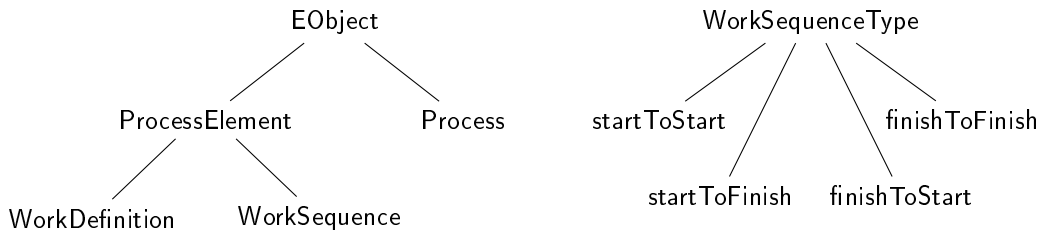
and containment associations, or references. In both cases, one-way references are represented by the addition of an argument to the data constructor corresponding to the target class. This argument has the same name of the reference. However, its type depends on the multiplicity of the reference: a single value indicates the type corresponding to the source class; a value range entails the definition of a new algebraic type and its data constructor denoting a variadic operator (through the use of **%oplist**). A bidirectional reference is equivalent to two one-way references in opposite directions.

5.4.2 Case study: SimplePDL to Petri net

The application of Tom-EMF is introduced through a case study on which we would like to transform a metamodel in the *Simple Process Description Language* (SimplePDL) formalism – presented in Chapter 2.2.2 of Benoît Combemale's thesis [Com08] – into a corresponding one in the Petri net formalism [Pet62, Mur89]. These metamodels were strongly inspired by the case study presented in the work of Benoît Combemale et al. [CGC+07].

The SimplePDL metamodel to be transformed (i.e. the source metamodel) is illustrated in Figure 5.2. It describes a process (Process) composed of process elements (ProcessElement) that can be either a work definition (WorkDefinition) or a work sequence (WorkSequence). Work definitions are activities to be performed during a process. Complementarily, work sequences define dependency relationship between two work definitions according to the value of the attribute linkType. For example, two work definitions predecessor and successor linked by a dependency relation of kind startToFinish state that successor is able to finish only when predecessor is already started (and respectively for startToStart, finishToStart and finishToFinish). Finally, a work definition can be described itself by a process through the notion of hierarchy.

The Tom mapping specifying a SimplePDL metamodel is automatically generated by Tom-EMF tool developed by Jean-Christophe Bach. It is initially explained in Section *Playing with EMF* of the Tom online documentation available at the Tom web site [12]. The Java classes and Tom algebraic data types used in the model transformation are organized in hierarchical structures and therefore consider subtyping as follows:



12. Reference available at http://tom.loria.fr . Visited on February 2012.

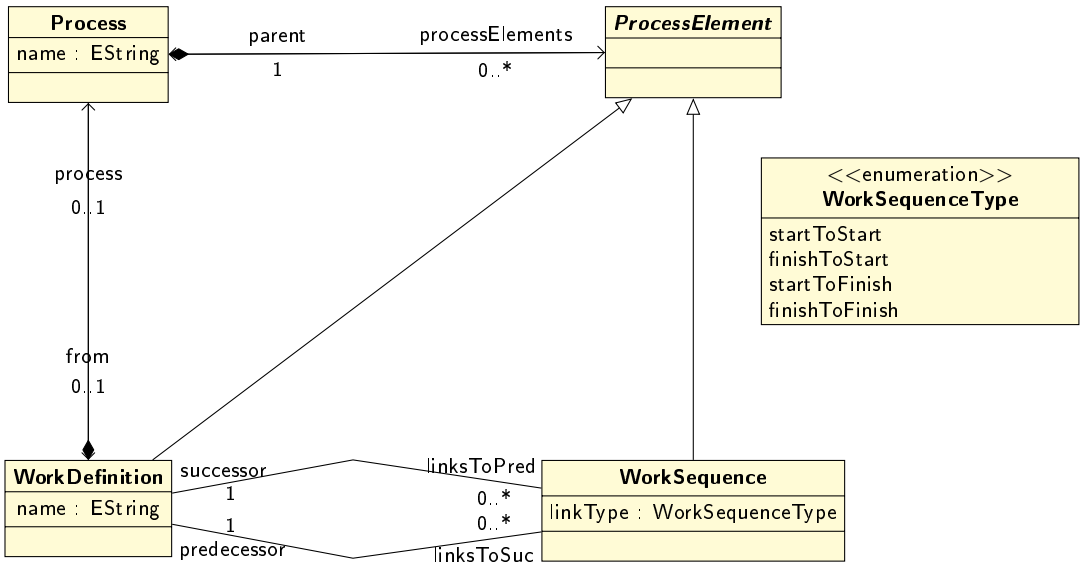
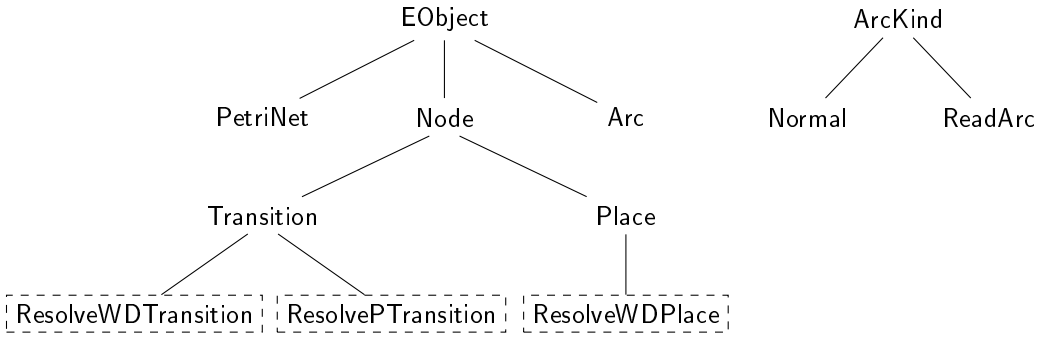


Figure 5.2: The source SimplePDL metamodel.

Note that data types prefixed by "E" (e.g. EObject, EString and EInt) are Ecore types usually corresponding to those Java types with similar non-prefixed names.

A possible metamodel for Petri nets is given in Figure 5.3 and is considered as the target metamodel of the transformation. A Petri net (PetriNet) is a composition of nodes (Node) that denote either a place (Place) or a transition (Transition). Nodes are linked by arcs (Arc) of kinds normal and read_arc. The attribute weight specifies the number of tokens consumed in source places (and respectively produced in target places). However, read-arcs only check whether the number of tokens of a place is in accordance with that provided by weight. A Petri net marking (marking) is defined by the number of tokens of places. Transitions express time intervals to be respected when a transition is enabled.

The Tom-EMF mapping generator also specifies Petri net metamodels through an hierarchical mapping:



In addition to the algebraic types representing the metamodel of Figure 5.3, data types ResolveWDTransition, ResolvePTransition and ResolveWDPlace were included

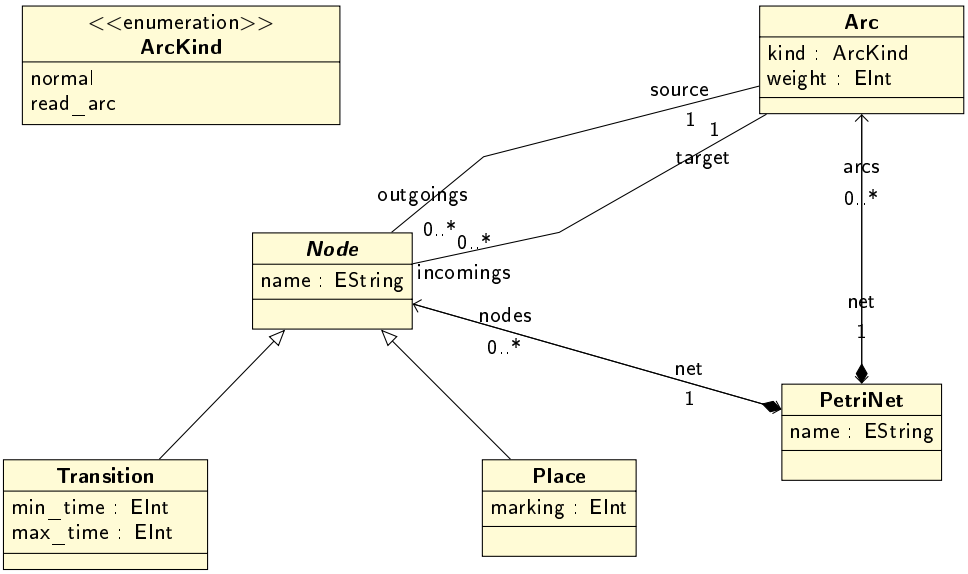


Figure 5.3: The target Petri net metamodel.

in the mapping. They extends Place and Transition implementations and allow to create nodes as temporary objects called Resolve objects. These intermediate nodes are used to transform the links between processes and activities, i.e. to store the sources and targets of arcs while waiting for the final resolution.

The model transformation approach adopted in this case study consists in decomposing the whole transformation of a process into several elementary transformations. They are disorderly executed and consequently can require the immediate creation and storage of Resolve objects to be definitively created by succeeding transformations. These elementary transformations are applied under the combination of strategy combinators Sequence and TopDown explained in Section 2.2.4. It defines a strategy transformer as follows:

```
Strategy transformer = 'Sequence(
    TopDown(Process2PetriNet(spdlModel)),
    TopDown(WorkDefinition2PetriNet(spdlModel)),
    TopDown(WorkSequence2PetriNet(spdlModel)));
```

where the term spdlModel represents a SimplePDL model to be transformed into a Petri net.

The transformation Process2PetriNet creates a Petri net every time it detects a process. Then, all nodes are stored on a hash map named NodeTable. In case of non-root processes, arcs are created between the process and the activity it describes. Likewise, the strategy WorkDefinition2PetriNet creates a Petri net to each work definition Wd found and adds the resulting pairs (Wd,WdPNet) to NodeTable. Then, the current activity is connected to its parent process, if any, by the creation of intermediate nodes ResolvePTransition. The transformation WorkSequence2PetriNet consists in a strategy that creates links between activities

of a same process.

In order to conclude the model transformation, temporary objects and partial results must be reconnected after the execution of all atomic transformations. Then, due to a storage table, Resolve objects are required to be replaced by "real" target objects existing in partial results. Therefore, the Petri net obtained from the application of the transformer strategy and composed of Resolve objects must replace them with places and transitions of the NodeTable. This is performed by the ResolveLinks strategy described by the following pseudocode:

```
GET PetriNet representing the initial SpdlModel
VISIT type Node WITH
    CTR ResolvePlace built from ResolveWDPlace constructor:
        Create Result of type Place
        SET Result to NodeTable[ResolvePlace.from] of name ResolvePlace.name
        RETURN Result
    ENDCTR

    CTR ResolveTransition built from ResolveWDTransition or ResolvePTransition constructors :
        Create Result of type Transition
        SET Result to NodeTable[ResolveTransition.from] of name ResolveTransition.name
        RETURN Result
    ENDCTR
ENDVISIT
```

We note the use of a non-standard construct VISIT-WITH-CTR-ENDCTR-ENDVISIT to indicate the type and data constructors of the terms considered by the strategy. The application of ResolveLinks to the Petri net is done under the TopDown combinator. Thanks to the definition of a hierarchical mapping, all terms whose type is a subtype of Node are distinctively traversed and transformed in a single **visit** construct.

5.5 Synthesis

Besides improved the type safety of the Tom language, the integration of our constraint-based type system into the Tom system increased its efficiency. In this chapter we initially introduced same simple examples in Tom to clearly indicate the power of type inference as well as its capacity to anticipate the detection of type errors signalized by the Java compiler or caught by a Java exceptions. In complement to the theoretical results previously presented, we provided a side-by-side comparison of programming in Tom with and without subtyping through a larger example involving symbolic expressions and statements. More examples available in Tom repository were considered in order to benchmark the performance of our type system against that existing in the beginning of this thesis.

A case study for the Tom-EMF tool was presented as an interesting and real application of subtyping in Tom programming. It consists in the use of Tom to perform model transformation between two EMF models. The metamodels

adopted in it are SimplePDL (source metamodel) and Petri net (target metamodel), both of them implemented in Tom through mappings organized in a hierarchical structure. The process of transformation is encoded by a combination of strategies. As a result of the expressiveness improvement arose from our type system, these strategies are able to consider the predefined type hierarchy when performing pattern matching over terms representing model elements.

Conclusion

Type systems are applied to computer programming as a formal method able to classify terms according to the kinds (i.e. sorts) of values they compute when executed. They aim to prove the absence, not of unexpected computed results, but of meaningless terms in regard to a type specification. This thesis is situated in the context of term rewriting embedded in object-oriented programming. Its purpose is to develop a safe type system compatible with the one of the host programming language. In particular, it features subtyping for support of associative pattern matching on algebraic data types with variadic constructors. In this work we specifically considered the Tom rewriting language and environment developed in the Pareo team-project.

Our first goal was to improve the safety of the Tom language using type checking and type inference as verification approaches applied by the typing mechanism. Then we followed the idea of John Mitchell [Mit84] about type inference as a form of type checking and then included constraints inside typing judgments. A preliminary difficulty arose over the definition of a constraint system whose constituting elements (i.e. equality constraints) are solved by unification and types are interpreted as unsorted terms. Such types suffice to distinguish between many-sorted terms built from identical variadic operators. We bypassed it by defining types as base sorts decorated with variadic function symbols. We also introduced type variables to represent uninterpreted types of the constraint-based type inference system as suggested in Benjamin Pierce's work [Pie02]. His work inspired the description of a type checking system as a proof calculus for typing judgments about type annotated expressions composing a Tom program. Another contribution was the formalization of the big-step operational semantics of the Tom language according to the formalism defined by Gilles Kahn [Kah87] and to the work of Didier Rémy [Rém02]. This was useful to prove the safety of our type system.

The enhancement of expressiveness in the pattern matching provided by Tom was our second and main goal. For this purpose, we intended to allow for type inclusion at the pattern level which led to a more permissive type compatibility. We decided on the Luca Cardelli's proposition [Car88] of a nominal type system considering subtyping as a partial order over types. Instead of considering only equality constraints, the type inference system generates a set of equality and subtyping (i.e. inequality) constraints to be simplified and subsequently solved. A major difficulty was to ensure that all constraint combinations are

handled by the rewrite rules of subtyping constraint resolution. Our starting point to resolve this problem was the simplification algorithm given by François Pottier [Pot01]. As him, we also split the simplification phase in subphases responsible for producing a smaller and simplified representation of all the solutions of a constraint set. However, contrary to Pottier and although we added rules for the generation of only one solution, we have not exhibited a class of constraint sets with solutions. But we have obtained interesting theoretical results as the proof of equivalence between the algorithmic approach of the type inference system and the declarative one of the type checking system enriched with a subsumption rule. The formalization of the operational semantics of the Tom language was extended to support subtyping and consequently allow the proof of the safety of the current type system.

In a practical viewpoint, after a successful prototyping phase, we focused on the implementation of the constraint-based type inference system in Tom. The Tom system has a pipeline structure in where an AST representing the input Tom program is sequentially transformed by each phase of compilation. It raised an initial difficulty concerning the identification of the subset of AST nodes to be considered by the typing rules. Although the addition of new data constructors to represent type constraints was impact-free for the other phases, it involved a refactoring of the Tom code so as to clean and separate the issues outside of the scope of the typer. What was implemented in the Tom compiler corresponds exactly to a derecursivated version of the classical syntactic unification algorithm and the type inference rules described in this thesis. This gives first of all a solid proven theoretical background to this part of the Tom system. Furthermore, this strong correspondence between the implementation and the theory allows us to back the claims of enhanced safety for the Tom system with the various proofs (soundness, completeness, preservation, progress, termination) we have written. All in all, we can say that we have both in theory and in practice obtained a more permissive type compatibility for Tom matching constructs while preserving type safety.

The additional cost of these features is not excessive. Experiments actually showed that time of type inference and total compilation of Tom programs can increase for larger applications, though we think it stays within acceptable limits. This increase is in fact rather expected as our typer performs type inference for a larger number of Tom constructs than the previously existing one. Nonetheless, we have tried to maintain the added time as low as possible through some shallow optimizations of the typer, for example with respect to the number of type variables and constraints generated in the type inference rules. We are quite confident that further work could unearth other deeper optimizations, both in scope and in time gained. In particular, we think that an in depth examination of the impact of strategies in the typer implementation could contribute to reduce typing time. A review of the data structures used could also shave some additional time off the typing phase. Finally, the subtyping constraint propagation and resolution algorithm could benefit from a second look.

Although being not directly related to our type system, the support of subtyping by the Gom tool constitutes a desirable facility for the definition of mappings organized in an inheritance structure. It would allow the standard utilization of Tom in where mappings are automatically generated by Gom. Accordingly, the type hierarchy defined by subtyping relations between algebraic data types would be automatically maintained by their mapping and concrete data types implementing it. We also envisage to provide a more direct use of subtyping in pattern matching through the introduction of *typed patterns* in Tom. They could be of the form `x : t` such that `x` is a pattern variable and `t` is an algebraic type pattern. This would discard the optional type annotation of the matching operator "<<" since a matching condition (i.e. match-equation) `x << t subj` is extensionally equivalent to `x : t << subj`. Thus, a typed pattern would match any subject whose value is of pattern type and could occur even in function arguments.

From a theoretical viewpoint, interesting improvements remain to be done in our subtyping-constraint-based type system. One could imagine a new garbage collection approach based on *polarities*. Adapting François Pottier's idea [Pot01], we should annotate all type variables with signals - and + representing respectively a type of an algebraic variable and a type of a concrete (i.e. non-algebraic) variable. This would allow superfluous constraints be eliminated before the phase of generation of a solution. Moreover, we expect to succeed in proving properties concerning the reliability of our type system. We particularly intend to show two theoretical results: *a*) our typing mechanism derives the least restrictive types that satisfy all the generated constraints, and *b*) the definition of the class of constraint sets for which our constraint solving algorithm is able to find a solution. Besides, a natural enhancement of our type system is the handling of new expressiveness features available within languages in where Tom is embedded. For instance, we could point out the *parametric polymorphism* as a very useful concept concerning both functional and object-oriented programming paradigms. Accordingly, it would enable common mappings and methods that include matching constructs to manipulate algebraic terms of different types in Tom.

Appendix A

Tom handwritten mapping

```
import tom.library.sl.*;

public class ExamplePeano{
  %include { sl.tom }
  //--------- Java classes ----------
  static class JNat { }
  static class JNatList { }

  static class Jzero extends JNat {
    public Jzero() {}
    public boolean equals(Object o) {
      if(o instanceof Jzero) {
        return true;
      }
      return false;
    }
  }

  static class Jsuc extends JNat {
    public JNat n;
    public Jsuc() { }
    public Jsuc(JNat n) { this.n = n; }
    public boolean equals(Object o) {
      if(o instanceof Jsuc) {
        Jsuc obj = (Jsuc) o;
        return n.equals(obj.n);
      }
      return false;
    }
  }

  static class Jplus extends JNat {
    public JNat n1;
    public JNat n2;
    public Jplus() { }
    public Jplus(JNat n1, JNat n2) {
      this.n1 = n1;
```

```
    this.n2 = n2;
  }
  public boolean equals(Object o) {
    if(o instanceof Jplus) {
      Jplus obj = (Jplus) o;
      return n1.equals(obj.n1) && n2.equals(obj.n2);
    }
    return false;
  }
}

static class JconcJNat extends JNatList {
  public JNat head;
  public JNatList tail;
  public JconcJNat() { head = null; tail = null; }
  public JconcJNat(JNat h, JNatList ntail) {
    head = h;
    tail = ntail;
  }
  public boolean isEmpty() {
    return (head == null && tail == null);
  }
  public boolean equals(Object o) {
    if (o instanceof JconcJNat) {
      JconcJNat obj = (JconcJNat) o;
      if (this.isEmpty() && obj.isEmpty()) {
        return true;
      } else if (!this.isEmpty() && !obj.isEmpty()) {
        return
          head.equals(obj.head) && tail.equals(obj.tail);
      }
    }
    return false;
  }
}
//--------- Tom mappings ----------
%typeterm Nat {
  implement    { JNat }
  is_sort(s)   { (s instanceof JNat) }
  equals(t1,t2) { (t1.equals(t2)) }
}

%typeterm NatList {
  implement    { JNatList }
  is_sort(s)   { (s instanceof JNatList) }
  equals(t1,t2) { (t1.equals(t2)) }
}

%op Nat zero() {
  is_fsym(s) { (s instanceof Jzero) }
  make()     { new Jzero() }
}

%op Nat suc(n:Nat) {
```

```
  is_fsym(s)    { (s instanceof Jsuc) }
  get_slot(n,s) { ((Jsuc)s).n }
  make(t0)      { new Jsuc(t0) }
}

%op Nat plus(n1:Nat,n2:Nat) {
  is_fsym(s)    { (s instanceof Jplus) }
  get_slot(n1,s) { ((Jplus)s).n1 }
  get_slot(n2,s) { ((Jplus)s).n2 }
  make(t0,t1)   { new Jplus(t0,t1) }
}

%oplist NatList concNat(Nat*) {
  is_fsym(s)        { (s instanceof JconcJNat) }
  get_head(l)       { ((JconcJNat)l).head }
  get_tail(l)       { ((JconcJNat)l). tail  }
  is_empty(l)       { ((JconcJNat)l).isEmpty() }
  make_empty()      { new JconcJNat() }
  make_insert(t,l)  { new JconcJNat(t,l) }
}
//-------------------------------
public static void main(String[] args) {
  ExamplePeano exPeano = new ExamplePeano();
  exPeano.run();
}

public void run() {
  JNat one = `suc(zero());
  JNat zero = `zero();
  JNat result = peanoPlus(one,zero);
  System.out.print("one + zero = " + printJNat(result) + "\n");

  JNatList nList = `concNat(zero(),suc(zero()),suc(suc(zero())));
   printSolutions(nList);

  try {
    JNat number = `plus(one,zero);
    Strategy addition = `addition();
    JNat reducedNumber1 = addition.visit(number, new LocalIntrospector());
    System.out.println("By addition: Term '" + printJNat(number) + "' reduces to '" +
        printJNat(reducedNumber1) +"'.");

    Strategy recursiveAddition = `TopDown(addition());
    JNat reducedNumber2 = recursiveAddition.visit(number, new LocalIntrospector());
    System.out.println("By recursiveAddition: Term '" + printJNat(number) + "' reduces to '" +
        printJNat(reducedNumber2) +"'.");
  } catch(VisitFailure e) {
    System.out.println("strategy failed");
  }

  int total = variadicPlus(nList);
  System.out.println("Total = " + total);

   printPositiveIntegers(nList);
```

```
}

public JNat peanoPlus(JNat t1, JNat t2) {
  %match(Nat t1, Nat t2) {
    zero(), x -> { return `x; }
    suc(y), x -> { return `suc(peanoPlus(y,x)); }
  }
  return null;
}

public int variadicPlus(JNatList nList) {
  %match {
    !concNat(x*,suc(y),z*) << nList || concNat() << nList -> {
      return 0;
    }
    concNat(x*,suc(y),z*) << nList -> {
      return variadicPlus(`concNat(y,x*,z*)) + 1;
    }
  }
  return -1;
}

public void printPositiveIntegers(JNatList nList) {
  int counter = 0;
  %match {
    concNat(x*,pnat@suc(y),z*) << NatList nList -> {
      counter++;
      System.out.println("Positive integer #" + counter + " = " + `pnat);
    }
  }
}

public String printFirstElement(JNat t) {
  %match {
    plus[n1=x] << t -> { return printJNat(`x); }
  }
  return "";
}

%strategy addition() extends Identity() {
  visit Nat {
    plus(zero(), x) -> { return `x; }
    plus(suc(y), x) -> { return `suc(plus(y,x)); }
  }
}

public void printSolutions(JNatList nList) {
  %match(nList) {
    concNat(x*,y*) -> {
      System.out.print("x = " + printJNatList(`x) + "\t\t");
      System.out.println("y = " + printJNatList(`y));
    }
  }
}
```

```
  public String printJNat(JNat n) {
    %match(n) {
      zero() -> { return "0"; }
      suc(x) -> { return ("suc(" + printJNat(‘x) + ")"); }
      plus(x1,x2) -> { return ("plus(" + printJNat(‘x1) + "," + printJNat(‘x2) + ")"); }
    }
    return "";
  }

  public String printJNatList(JNatList nList) {
    String result = "(";
    %match(nList) {
      concNat(x) -> { return (result + printJNat(‘x) + ")"); }
      concNat(head,tail*) -> { result += (printJNat(‘head) + "," + printJNatList(‘ tail )); }
    }
    // CASE concNat()
    return ( result += ")");
  }
}
```

Appendix B

Notational conventions

B.1 Metavariable names

Name	Usage
f	syntactic operators
v	variadic operators
g	both syntactic or variadic operators
t, u	terms
x, y, z	term variables
s	sorts
f, v, h, ?	sort decorations
τ	types
s^h,wt	base types
α	type variables
φ	typing judgments
π	typing derivations
e	Tom expressions
$\dot{v}$	values
$\dot{u}$	groud terms as values
ρ	environments
$\bar{\rho}$	collection of environments
σ, δ, ϕ	substitutions
Γ	contexts
γ	pairs (variable,type) and (operator,rank)

B.2 Rule naming convention

Prefix	Usage
E-	evaluation with $\Vdash$
T-	typing with $\vdash$
CT-	constraint typing with $\vdash_{ct}$

Index

Bibliography

[AI04] Freddy Allilaire and Tarik Idrissi. Adt: Eclipse development tools for atl. In *Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2)*, pages 171–178. Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK, 2004. 181

[AW92] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *LICS*, pages 329–340, 1992. 3, 49, 140

[Bal09] Emilie Balland. *Conception d'un langage dédié à l'analyse et la transformation de programmes*. PhD thesis, Université Henri Poincaré - Nancy I, 03 2009. 84

[BBB+11] Jean-Christophe Bach, Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, Antoine Reilles, and Cláudia Tavares. Tom-2.8 documentation, 2011. 1, 47, 71, 81

[BBK+07] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In *Conference on Rewriting Techniques and Applications - RTA'07 Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *LNCS*, pages 36–47, Paris/France France, 06 2007. Springer-Verlag. 1, 47, 71

[BBR96] Narjes Berregeb, Adel Bouhoula, and Michaël Rusinowitch. Automated verification by induction with associative-commutative operators. In *CAV*, pages 220–231, 1996. 65

[BCD+06] Peter Borovanský, Horatiu Cirstea, Eric Deplagne, Hubert Dubois, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN V3.7 User Manual, 2006. 4, 50, 92

[BJM00] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theor. Comput. Sci.*, 236:35–132, April 2000. 89, 141

[BKK+97] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. Elan: A logical framework based on computational systems. Elsevier, 1997. 93

[BKK+98] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of elan. 15, 1998. 4, 50, 92

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998. 57

[Bon04] Richard Bonichon. Tamed: A tableau method for deduction modulo. In *IJCAR*, pages 445–459, 2004. 72

[Bor98] Peter Borovanský. *Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, octobre 1998. 93

[Bra10] Paul Brauner. *Fondements et mise en œuvre de la Super Déduction Modulo.* PhD thesis, Université Henri Poincaré - Nancy I, 06 2010. 72

[Bur09] Guillaume Burel. *Bonnes démonstrations en déduction modulo.* PhD thesis, Université Henri Poincaré (Nancy 1), 2009. 72

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66. 2, 41, 48, 140, 189

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. 4, 50, 89, 141

[CF58] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, 1958. Second printing 1968. 89

[CGC+07] Benoît Combemale, Pierre-Loïc Garoche, Xavier Crégut, Xavier Thirioux, and François Vernadat. Towards a formal verification of process model's properties simplepdl and tocl case study. In *ICEIS (3)*, pages 80–89, 2007. 183

[CK01a] Horatiu Cirstea and Claude Kirchner. The rewriting calculus - part i. *Logic Journal of the IGPL*, 9(3):339–375, 2001. 98

[CK01b] Horatiu Cirstea and Claude Kirchner. The rewriting calculus - part ii. *Logic Journal of the IGPL*, 9(3):377–410, 2001. 98

[CKKM10] Horatiu Cirstea, Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-patterns for rule-based languages. *J. Symb. Comput.*, 45:523–550, May 2010. 80

[CLW03] Horatiu Cirstea, Luigi Liquori, and Benjamin Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *TYPES*, pages 147–161, 2003. 98

[Com08] Benoit Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle – Application à l'ingénierie des procédés*. PhD thesis, Institut National Polytechnique, Université de Toulouse, July 2008. in french. 183

[Cre06] Vincent Cremet. *Foundations for SCALA*. PhD thesis, Lausanne, 2006. 4, 50, 90, 142

[DFF+10] Zoé Drey, Cyril Faucher, Franck Fleurey, Vincent Mahé, and Didier Vojtisek. *Kermeta Language, Reference Manual*, 2010. 181

[DOS88] Nachum Dershowitz, Mitsuhiro Okada, and G. Sivakumar. *Canonical conditional rewrite systems*, volume 310, pages 538–549. Springer Berlin / Heidelberg, 1988. 70

[Emi07] Burak Emir. *Object-oriented pattern matching*. PhD thesis, Lausanne, 2007. 91, 143

[FGJM85] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of obj2. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '85, pages 52–66, New York, NY, USA, 1985. ACM. 89

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *ESOP*, pages 94–114, 1988. 3, 49, 140

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. Edinburgh lcf. Lecture Notes in Computer Science, 1979. 140

[GWM+93] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean P. Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge University Press, 1993. 89, 92, 141

[Hil01] E. J. Friedman Hill. Jess, the Java Expert System Shell. Technical report, SANDIA National Laboratories, 2001. 89

[Hil03] E. J. Friedman Hill. Jess, the Expert System Shell for the java Platform. Technical report, SANDIA National Laboratories, 2003. 89

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of The American Mathematical Society*, 1969. 2, 32, 48, 89, 130

[HKK98] Claus Hintermeier, Claude Kirchner, and Hélène Kirchner. Dynamically typed computations for order-sorted equational presentations. *J. Symb. Comput.*, 25(4):455–526, 1998. 140

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua - An Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652, 1996. 88

[JBF11] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, pages 201–221, Berlin, Heidelberg, 2011. Springer-Verlag. 181

[JK86] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15:1155–1194, November 1986. 5, 50, 68

[JK91] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991. 19, 100

[JKKM92] J.-P. Jouannaud, Claude Kirchner, Hélène Kirchner, and A. Mégrelis. Programming with equalities, subsorts, overloading and parameterization in OBJ. *Journal of Logic Programming*, 12(3):257–280, February 1992. 141

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, LFP '92, pages 193–204, New York, NY, USA, 1992. ACM. 3, 49, 140

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '87, pages 22–39, London, UK, 1987. Springer-Verlag. 6, 17, 41, 51, 99, 189

[Kes91] Delia Kesner. *Pattern matching in order-sorted languages*. Rapports de recherche. Université Paris-Sud, Centre d'Orsay, Laboratoire de recherche en Informatique, 1991. 6, 21, 51, 146

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag. 89

[Kir90] Claude Kirchner, editor. *Unification*. Academic Press, London, UK, 1990. 59

[KK06] Claude Kirchner and Hélène Kirchner. Rewriting solving proving, 2006. 57, 59, 65, 66, 68

[KKM07] Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 110–124, Berlin, Heidelberg, 2007. Springer-Verlag. 80

[KKM08] Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Language and automata theory and applications. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Proceedings of the 2nd International Conference on Language and Automata Theory and Applications*, LATA'08, pages 275–286, Berlin, Heidelberg, 2008. Springer-Verlag. 80

[KMT09] Claude Kirchner, Pierre-Etienne Moreau, and Cláudia Tavares. A type system for tom. In *RULE*, pages 51–63, 2009. 139

[Kop08] Radu Kopetz. *Contraintes d'anti-filtrage et programmation par réécriture*. PhD thesis, Institut National Polytechnique de Lorraine - INPL, October 2008. 19, 21, 80, 100, 102

[Mar93] Claude Marché. *Réécriture modulo une théorie présentée par un système convergent et décidabilité des problèmes du mot dans certaines classes de théories équationnelles*. Thèse de doctorat, Université Paris-Sud, Orsay, France, October 1993. 65

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96:73–155, April 1992. 89, 92

[Mes97] José Meseguer. Membership algebra as a logical framework for equational specification. In *Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques*, WADT '97, pages 18–61, London, UK, 1997. Springer-Verlag. 89, 141

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 2, 32, 48, 89, 130

[Mit84] John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 175–185, New York, NY, USA, 1984. ACM. 2, 41, 48, 140, 189

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 61–76, Berlin, Heidelberg, 2003. Springer-Verlag. 71

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989. 183

[Nie02] Pat Niemeyer. The BeanShell User Manual - version 1.3, 2002. 88

[Ode11] Martin Odersky. *The Scala Language Specification Version 2.9*. Lausanne, Switzerland, May 2011. 4, 50, 90, 142

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962. 183

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 27, 41, 57, 121, 189

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981. 19, 100

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Program*, 60-61:17–139, 2004. 19, 100

[Pot98a] François Pottier. *Synthèse de types en présence de sous-typage: de la théorie à la pratique*. PhD thesis, Université Paris 7, July 1998. 35, 160

[Pot98b] François Pottier. Type inference in the presence of subtyping: from theory to practice. Research Report 3483, INRIA, September 1998. 35, 160

[Pot01] François Pottier. Simplifying subtyping constraints: a theory. *Inf. Comput.*, 170(2):153–183, 2001. 3, 35, 42, 43, 49, 140, 160, 190, 191

[PS81] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28(2):233–264, 1981. 5, 50, 68

[Ré02] Didier Rémy. Using, understanding, and unraveling the ocaml language. from practice to theory and vice versa. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 413–536, London, UK, 2002. Springer-Verlag. 6, 17, 41, 51, 99, 189

[Rei07] Antoine Reilles. Canonical abstract syntax trees. *Electronic Notes in Theoretical Computer Science*, 176(4):165 – 179, 2007. Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006). 10, 75

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999. 182

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. 181, 182

[Tav11] Cláudia Tavares. A type system for embedded rewriting languages with associative pattern matching: from theory to practice. Research report, November 2011. 139

[TS96] Valery Trifonov and Scott F. Smith. Subtyping constrained types. In *SAS*, pages 349–365, 1996. 3, 49, 140

[VBT98] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 13–26, New York, NY, USA, 1998. ACM. 12, 82

[Vit94] Marian Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, octobre 1994. 93

Résumé

Dans le domaine du génie logiciel, les systèmes de types sont souvent considérés pour la prévention de l'occurrence de termes dénués de sens par rapport à une spécification des types. Cette thèse se situe dans le contexte de la réécriture de termes embarquée dans la programmation orientée objet. Elle vise à développer un système de types avec sous-typage pour le support du filtrage de motifs associatif sur des termes algébriques construits sur des opérateurs variadiq langage de réécriture Tom qui fournit des constructions de filtrage de motifs et des stratégies de réécriture à des langages généralistes comme Java.

Nous décrivons l'évaluation de code Tom à travers la définition de la sémantique opérationnelle de ce langage en tant qu'élément essentiel de la preuve de la sûreté du système de types. Celui-ci inclut la vérification de types ainsi que l'inférence de types à base de contraintes. Le langage de contraintes est composé d'une part, de contraintes d'égalité, résolues par unification, d'autre part, de contraintes de sous-typage, résolues par la combinaison de phases de simplification, de génération d'une solution et de ramassage de miettes. Le système de types a été intégré au langage Tom, ce qui permet une plus forte expressivité et plus de sûreté a fin d'assurer que les transformations décrites par des règles de réécriture préservent le type des termes.

Mots-clés: systèmes de types, sous-typage, filtrage de motifs, réécriture, contraintes de type.

Abstract

In software engineering, type systems are often considered in order to prevent the occurrence of meaningless terms in regard to a type specification. This thesis is situated in the context of term rewriting embedded in object-oriented programming and aims to develop a safe type system featuring subtyping for the support of associative pattern matching on algebraic terms built from variadic operators. In this work we consider the Tom rewriting language that provides associative pattern matching constructs and rewrite strategies for Java.

We describe Tom code evaluation through the definition of the operational semantics of the Tom language as an essential element to show that the type system is safe. The type system includes type checking and constraint-based type inference. The constraint language is composed of equality constraints solved by unification and subtyping constraints solved by a combination of simplification, generation of solution and garbage collecting. The type system was integrated in Tom which provides both stronger expressiveness and more safety able to ensure that the transformations described by rewrite rules preserve the type of terms.

Keywords: type systems, subtyping, pattern matching, rewriting, type constraints.