Cartesian Programming

Habilitation Thesis
University of Grenoble, France

John Plaice
`plaice@cse.unsw.edu.au`

Technical Report
UNSW-CSE-TR-1101
January 2011

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

La programmation cartésienne

John PLAICE

Thèse présentée pour l'obtention d'une
Habilitation à Diriger des Recherches
Université de Grenoble, France

soutenue le 20 décembre 2010 devant la commission d'examen

MM.	Joseph SIFAKIS	**Président**
	Albert BENVENISTE	**Examinateurs**
	Paul CASPI	
	Nicolas HALBWACHS	
	Satnam SINGH	**Rapporteurs**
	Jean-Pierre TALPIN	
	William W. WADGE	

Abstract

We present a new form of declarative programming inspired by the Cartesian coordinate system. This *Cartesian programming*, illustrated by the *TransLucid* language, assumes that all programmed entities vary with respect to all possible *dimensions*, or degrees of freedom. This model is immediately applicable to areas of science, engineering and business in which working with multiple dimensions is common. It is also well suited to specification and programming in situations where a problem is not fully understood, and, with refinement, more parameters will need to be taken into consideration as time progresses.

In the Cartesian model, these dimensions include, for all entities, all possible parameters, be these explicit or implicit, visible or hidden. As a result, defining the aggregate semantics of an entire system is simplified, much as the use of the universal relation simplifies the semantics of a relational database system. Evolution through time is handled through the use of a special time dimension that does not allow access to the future.

In TransLucid, any atomic value may be used as a dimension. A *context* maps each dimension to its corresponding *ordinate*. A *context delta* is the partial equivalent. An expression is evaluated in a given context, and this context may be queried, dimension by dimension, or perturbed by a context delta.

A variable in TransLucid may have several definitions and, given a current context, the *bestfit* (most specific) definitions with respect to that context are chosen and evaluated separately, and the results are combined together. The set of definitions for a variable define a *hyperdaton*, which can be understood as an arbitrary-dimensional array of arbitrary extent.

Functional abstraction in TransLucid requires two kinds of parameters: *value* parameters, with call-by-value semantics, are used to pass dimensions and constants; *named* parameters, with call-by-name semantics, are used to pass hyperdatons. Where clauses, used for local definitions, define both new variables and new dimensions of variance.

This thesis presents the full development of Cartesian programming and of TransLucid, complete with a historical overview leading to their conception. Both the denotational and operational semantics are presented, as is the implementation, designed as a library. One important result is that the operational semantics requires only the evaluation and caching of *relevant* dimensions, thereby ensuring that space usage is kept to a minimum. Two applications using the TransLucid library are presented, one a standalone interpreter, the other an interactive code browser and hyperdaton visualizer.

The set of equations defining a TransLucid system can vary over time, a special dimension. At each instant, the set of equations may be modified, but in so doing can only affect the present and future of the system. Timing semantics is always synchronous. There are several possible ways for multiple TransLucid systems to interact.

The caching mechanism provided in the operational semantics allows for the efficient implementation of systems whose calling structure is highly irregular. For more regular structures, it is possible to create even more efficient bottom-up solutions, in which recursive instantiations of functions are eliminated, with clear bounds on memory usage and computation.

Cartesian programming is not just designed as a standalone paradigm, but as a means of better understanding other paradigms. We examine imperative programming and side-effects, and show that these, under certain conditions, can be translated into TransLucid, thereby allowing the design of new imperative constructs in the original language.

Résumé

Nous présentons une nouvelle forme de programmation déclarative inspirée par le système de coordonnées cartésien. Cette *programmation cartésienne*, illustrée par le langage *TransLucid*, suppose que toute entité programmée varie par rapport à toutes les *dimensions*, ou degrés de liberté. Ce modèle est immédiatement utilisable dans les domaines de la science, du génie ou des affaires dans lesquels travailler avec de multiples dimensions est courant. Le modèle est aussi bien appliqué à la spécification et la programmation dans des situations où un problème n'est pas entièrement compris et où de nouveaux paramètres seront pris en compte avec le fil du temps.

Dans le modèle cartésien, ces dimensions incluent, pour toute entité, tous les paramètres possibles, que ces derniers soient explicites ou implicites, visibles ou cachés. Il en suit que définir la sémantique collective d'un système complet est simplifié, de la même manière que l'utilisation de la relation universelle simplifie la sémantique d'un système de bases de données relationnel. L'évolution dans le temps se fait par moyen d'une dimension spéciale temporelle qui ne permet pas d'accès au futur.

En TransLucid, toute valeur atomique peut être utilisée comme dimension. Un *contexte* est une fonction totale portant chaque dimension à son *ordonnée* correspondante. Un *delta de contextes* est l'équivalent partiel. Une expression est évaluée dans un contexte donné, et ce contexte peut être interrogé, dimension par dimension, ou perturbé par un delta de contextes.

Une variable en TransLucid peut avoir plusieurs définitions et, étant donné un contexte courant, les définitions «bestfit» (les plus spécifiques) par rapport au contexte sont choisies et évaluées séparément, et les résultats sont rassemblés. L'ensemble des définitions d'une variable définit un *hyperdaton*, qui peut être percu comme un tableau de taille et de dimensionalité arbitraires.

L'abstraction fonctionnelle en TransLucid nécessite deux types de paramètres : les paramètres de *valeur*, avec sémantique «call-by-value», sont utilisés pour transmettre des dimensions et des constantes ; les paramètres *nommés*, avec sémantique «call-by-name», sont utilisés pour transmettre des hyperdatons. Les clauses `where`, utilisées pour les définitions locales, définissent à la fois de nouvelles variables et de nouvelles dimensions de variance.

La thèse présente le développement de la programmation cartésienne et de TransLucid, avec un survol historique menant à leur conception. Les sémantiques dénotationnelle et opérationnelle sont présentées, ainsi que l'est la réalisation, conçue comme une bibliothèque. Un résultat important est que la sémantique opérationnelle ne nécessite que l'évaluation et la mémorisation de dimensions *pertinentes*, minimisant ainsi l'utilisation d'espace. Deux logiciels utilisant la bibliothèque TransLucid sont présentés, un interprète à boucle textuelle, et un navigateur de code et d'hyperdatons.

Un ensemble d'équations définissant un système TransLucid peut varier dans le temps, une dimension spéciale. À chaque instant, l'ensemble d'équations peut être modifié, mais en ce faisant ne peut influencer que le présent et le futur du système. La sémantique temporelle est toujours synchrone. L'interaction de multiples systèmes TransLucid peut se faire de plusieurs manières.

Le mécanisme de mémorisation fourni dans la sémantique opérationnelle permet la réalisation efficace de systèmes dont l'arborescence des appels est très irrégulière. Pour des cas plus réguliers, il est possible de créer des solutions plus efficaces par le bas, dans lesquelles des appels récursifs de fonctions sont éliminés, avec des bornes claires sur l'utilisation de la mémoire et du calcul.

La programmation cartésienne n'est pas seulement conçue comme un paradigme à part, mais aussi comme une méthode pour mieux comprendre d'autres paradigmes. Nous examinons la programmation impérative et les effets de bord et démontrons que, sous certaines conditions, ceux-ci peuvent être traduits en TransLucid, permettant ainsi la conception de nouvelles constructions impératives dans le langage originel.

Acknowledgements

I wish to thank the three *rapporteurs* of my work, Satnam Singh, Jean-Pierre Talpin and William Wadge, for having accepted to evaluate this thesis with a very short turnaround time. I wish to thank the other examiners of the jury, Albert Benveniste, Paul Caspi, Nicolas Halbwachs, and Joseph Sifakis for having taken the time from their Christmas holidays so that they could attend my defense. For me, it is a great honor to be able to present my work to such a distinguished jury.

The present work is the result of many years of reflection, of individual steps forward followed by multiple steps backwards, of attempted semantics, of software development, of experiments, with successes and failures. The advances presented here would never have taken place without the support of a number of people and organizations.

First, I would like to thank the School of Computer Science and Engineering at The University of South Wales in Sydney, Australia for providing me with a base where I was allowed to do things my unorthodox way, without ever being told that I had to do things differently. My thanks go to my Head of School for a number of years, Paul Compton.

Second, I wrote this thesis while on sabbatical from UNSW, for 9 months at the University of Neuchâtel in Switzerland and 3 months at the Tata Institute of Fundamental Research in Mumbai, India. I received support from the Swiss National Foundation and from TIFR. For this I am grateful. My thanks go to Peter Kropf, Pascal Felber and Jacques Savoy in Neuchâtel and to R. K. Shyamasundar, Paritosh K. Pandya and N. Raja in Mumbai for interesting discussions.

The intuitions behind the ideas in this work go back to a series of conversations that took place between 1999 and 2004: with Peter Kropf, telling me that it was time for me to start working on something significant; with William Wadge, with whom I discovered the pervasive nature of the intensional context; with Paul Swoboda, who did his PhD under my supervision, for helping me invent the æther as a shared context acting as a publish-and-subscribe system; and with David Parnas, who claimed that I was simply reinventing distributed, global variables.

The experiments took place while working with my students, both PhD and undergraduate. Paul Swoboda wrote an impressive suite of software used to experiment with intensional systems, while Gabriel Ditu wrote the initial presentation for TransLucid. Several TransLucid interpreters were written by undergraduates, including Myatt Khine Oo, Toby Rahilly and Jarryd Beck.

Finally, I wish to thank my wife, Blanca Lucía Mancilla, our two daughters Carolina Luciana Plaice and Alejandra Lucía Plaice, and my son, Cyril Lucien Bouland, for encouraging me this past year to finish this work. Blanca and I have been married 15 years and she has been working with me for the last eight, which includes listening to my crazy ideas in the middle of the night.

Contents

Sommaire

Introduction

Cette thèse examine plusieurs aspects de l'informatique à partir d'une perspective cartésienne, essayant ainsi de fournir un cadre unifié basé sur le système de coordonnées cartésien. Au cœur de ce travail est l'introduction de la Programmation Cartésienne, dans laquelle toute entité est sensée varier par rapport à un contexte multidimensionnel, et le langage de programmation déclarative TransLucid, réalisant les idées de la programmation cartésienne.

L'intuition clé est que même les problèmes les plus simples sont multidimensionnels, et contiennent beaucoup de paramètres que l'on pourrait supposer initialement. Par exemple, si nous considérons le temps que devrait prendre un corps en chute libre pour atteindre le sol, nous débuterons avec la loi de Newton, en supposant que G soit constant. Mais ce problème est en réalité très complexe, puisqu'il y a plein de paramètres : la valeur exacte de G dans le lieu spécifique, l'humidité dans l'air, les vents actuels, la densité de l'object, la forme de l'objet, les propriétés magnétiques à la fois de l'objet et du lieu, et ainsi de suite. Et, bien sûr, si c'est un oiseau ou un avion, tout change.

Pour ce genre de problème, il est typique de programmer en supposant un ensemble limité de paramètres, puis d'ajouter au fur et à mesure des paramètres supplémentaires afin de prendre en compte un ensemble de plus en plus grand de situations ou d'atteindre un niveau de précision ou de contrôle sur le problème. Pendant la programmation, il s'avérera que dans un ensemble donné de situations, un ensemble de paramètres sera prépondérant, tandis que dans un autre ensemble donné de situations, d'autres paramètres seront importants. Dans certaines situations exceptionnelles, toute la programmation précédente devra être mise de côté et un code tout à fait nouveau devra être écrit.

Si nous considérons une autre branche de l'informatique, telle que la visualisation d'informations géographiques, il y aura aussi de nombreux paramètres : la projection géographique, les paramètres de la projection spécifique choisie, la portion du globe qui est représentée, la résolution de l'écran, la résolution des bases de données choisies, et ainsi de suite. Même le texte lui-même a de nombreux paramètres : codage de caractères, écriture, langue, utilisation de translitération, fonte, style, etc.

Dans la programmation cartésienne, il est supposé que *toutes* les entités programmées varient, dès le début, dans *toutes les dimensions possibles*. Initialement, la programmation de ces entités par moyen d'équations se fait par rapport à un nombre restreint de dimensions ; avec le progrès du temps, de plus en plus d'équations sont ajoutées, de plus en plus spécifiques pour des situations données. L'idée est que quand une valeur est requise dans un contexte donné, les définitions les plus pertinentes de cette variable, par rapport à ce contexte, sont choisies. Puis, si certaines conditions sont exceptionnelles, ces conditions peuvent être spécifiées dans un nouvel ensemble d'équations, et ces équations seront choisies quand elles seront nécessaires. Tout cela sans changer les équations précédentes, à moins que celles-ci se sont avérées être erronées, nécessitant ainsi d'être remplacées.

La programmation cartésienne n'est pas venue au monde par un grand «Fiat Lux !» Donc ce document a comme objectif de présenter les développements clé qui ont mené à la conception-même de la programmation cartésienne : la conception, la sémantique et la réalisation du langage TransLucid, et les applications actuelles développées à l'aide de TransLucid.

Le corps principal de la thèse est divisé en trois parties. La partie I situe la recherche en programmation cartésienne dans son contexte historique, soulignant les développements clé qui ont précédé l'introduction du terme «Programmation cartésienne». La partie II présente le language TransLucid, sa sémantique, à la fois dénotationnelle et opérationnelle, et la réalisation actuelle. La partie III est utilisée pour l'exploration de plusieurs sujets en informatique à partir de la perspective cartésienne : un interprète TransLucid indépendant, un navigateur de code TransLucid et d'hyperdatons, la réalisation efficace de fonctions récursives, le flux de contrôle et les effets de bord, et les paradigmes de programmation et les formes de conception.

La partie I, l'histoire, comprend deux chapitres physiques, mais il existe trois fils entrelacés : la *programmation intensionnelle*, la *programmation synchrone* et le *versionnement par mondes possibles*.

En 1984, quand l'auteur était toujours étudiant de premier cycle à l'Université de Waterloo, il a suivi un cours sur la logique intensionnelle de Richard Montague, qui a utilisé la sémantique des mondes possibles et les structures de Kripke afin de donner une sémantique à des fragments de l'anglais. Trois ans plus tard, en écrivant sa thèse de doctorat à l'Institut National Polytechnique de Grenoble, l'auteur a été surpris de tomber sur un article écrit par Anthony Faustini et William Wadge, intitulé «Intensional Programming», qui démontrait comment le langage Lucid de l'époque pouvait être compris en utilisant la sémantique des mondes possibles.

Le chapitre 1 présente une histoire du développement des versions successives du langage Lucid, à partir du Lucid originel de 1975 jusqu'aux propositions initiales de TransLucid. L'évolution du langage a été accompagnée par une évolution des mots utilisés : programmation flux de données, programmation intensionnelle, programmation indexicale et programmation cartésienne. On peut résumer ce développement comme un processus de «libération des dimensions». Le premier article sur Lucid a utilisé la multidimensionnalité, mais seulement une dimension pouvait être manipulée à la fois, les dimensions devaient apparaître dans un ordre prédéterminé, et les dimensions n'étaient pas accessibles au programmeur. Le TransLucid présenté dans cette thèse considère toutes les dimensions comme étant égales et permet à toutes les dimensions d'être manipulées directement et simultanément.

Dans la programmation synchrone, l'hypothèse clé est que les entrées ne changent pas pendant que les sorties sont calculées ; si ceci s'avère être vrai, on peut supposer que les entrées et les sorties sont simultanées. Le language LUSTRE développé par Nicolas Halbwachs et Paul Caspi a fourni une sémantique synchrone à un Lucid unidimensionnel, avec l'intuition que les i-èmes entrées de deux flux partageant la même horloge soient générées dans le même instant.

L'histoire de la programmation synchrone a déjà été écrite [9]. L'auteur a joué un rôle important au début, en écrivant dans le contexte de son doctorat les premières sémantiques dénotationnelle et opérationnelle et le premier compilateur pour LUSTRE. Ce language est le noyau du logiciel de programmation Scade, vendu actuellement par Esterel Technologies et utilisé pour la programmation de l'avionique Airbus, parmi tant d'autres choses.

Le chapitre 2 présente une histoire du développement du versionnement par les mondes possibles. (Appelé au début le *versionnement intensionnel*, il a été renommé puisqu'il existait déjà une autre utilisation de cette expression.) Ici, la sémantique des mondes possibles de la logique intensionelle a été appliquée à la *structure* des programmes, plutôt qu'à leur comportement. Proposé à l'origine pour des fins de contrôle de versions de logiciels, le versionnement par mondes possibles a été utilisé pour développer des formes versionnées de plusieurs types différents de logiciel : sites sur la Toile, langages de programmation orientés-objet, processus Linux, etc.

Avec le développement de ces expériences, de nouvelles intuitions sont arrivées. Premièrement, on peut donner au contexte multidimensionnel fournissant l'index aux entités versionnées une interprétation *physique* : une entité est plongée dans un contexte *pénétrant*, donc si le contexte est changé alors chaque aspect de l'entité est informée par ce changement de contexte et peut s'adapter en conséquence. La deuxième intuition est, étant donné ce contexte physique, que plus d'une entité peut partager ce même contexte, et que ces entités peuvent se communiquer entre elles par la diffusion, simplement en changeant ce contexte partagé, appelé maintenant *éther*.

La programmation cartésienne incorpore à la fois la programmation intensionnelle et le versionnement par mondes possibles. La structure et le comportement d'une entité, representés par des équations, varient selon le contexte. Cependant, une dimension joue le rôle primordial : le temps. Avec le temps, tout dans un système peut évoluer, même l'ensemble des dimensions disponibles pour la spécification de variance. Étant donné l'expérience de l'auteur à Grenoble avec la programmation synchrone, retourner à Grenoble pour présenter sa thèse d'habilitation semble naturel.

La partie II présente le langage TransLucid à partir de ses fondements. Les trois chapitres contiennent les détails techniques clé nécessaires pour le développement correct du langage.

Le chapitre 3 introduit le language avec une présentation intuitive d'un grand nombre d'exemples. Le chapitre peut être compris par quelqu'un avec des connaissances de base en mathématiques d'école secondaire. Il débute avec un récapitulatif du système de coordonnées cartésien, et peut être lu sans examiner les détails sémantiques.

Quand une expression dans un programme cartésien est exécuté, il y a toujours un contexte courant, c-à-d une fonction à partir d'un ensemble arbitraire (potentiellement infini) de dimensions vers des valeurs. Durant l'évaluation d'une expression, le contexte peut être perturbé en changeant les valeurs de certaines des dimensions ou peut être questionné, dimension par dimension. Quand une variable est rencontrée, il pourrait y avoir plusieurs définitions applicables au contexte actuel. Les définitions les plus pertinentes sont choisies, nécessairement par rapport à un nombre fini de dimensions, et l'évaluation continue.

Les fonctions peuvent prendre deux genres de paramètres : les paramètres par valeur, utilisés pour les dimensions et les constantes, sont évalués avant l'entrée dans le corps de la fonction ; les paramètres par nom, utilisés pour les hyperdatons, sont évalués sur demande dans le corps de la fonction.

Le chapitre 4 présente la sémantique de TransLucid en trois parties. La première est la sémantique dénotationnelle, qui est donnée par la sémantique du plus petit point fixe sur des structures de dimensionnalité arbitraire définies à partir d'ensembles d'équations récursives, c-à-d avec une évaluation par le bas. La seconde est une sémantique opérationnelle dirigée par la demande, qui n'évalue que les expressions qui doivent être évaluées. La troisième est une sémantique opérationnelle multi-fils dirigée par la demande, utilisant une cache dépendant du contexte. L'évaluateur et la cache jouent un jeu, s'assurant que les informations dans la cache ne font référence qu'aux dimensions actuellement rencontrées pendant l'évaluation des expressions.

Le chapitre 5 transforme le langage Core TransLucid en un vrai environnement de programmation, utilisable comme un langage de coordination sur `C++`. Un système TransLucid est un système réactif qui, à chaque instant, a un ensemble courant d'équations, d'entrées, de sorties et de demandes. Quand l'évaluation d'un instant débute, toutes les demandes pertinentes sont évaluées, avec leurs résultats placés dans les sorties appropriées.

La structure de données clé est l'*hyperdaton*, un foncteur `C++` qui retourne une valeur quand elle est fournie un contexte en entrée. Toutes les entités parsées, constantes, variables, expressions et équations, sont des sousclasses de la classe hyperdaton. Cette classe peut aussi être sousclassée afin de permettre la création d'objects utilisables à la fois en `C++` et en TransLucid.

Des moyens sont aussi fournis pour l'utilisation de types de données et d'opérations définis par l'utilisateur, à la fois au niveau sémantique qu'au niveau syntaxique, et le parseur très flexible permet l'introduction de nouveaux opérateurs préfixes, postfixes et infixes.

La partie III est intitulée «Exploration de l'espace cartésien».

Le chapitre 6, intitulé «Le temps pour les applications», présente deux applications qui utilisent la réalisation de la bibliothèque TransLucid. Le première est une boucle interactive textuelle, dans laquelle le système des équations est un système réactif synchrone : à chaque instant, des équations peuvent être ajoutées, supprimées ou remplacées, affectant seulement le comportement futur du système. La seconde est un système graphique permettant la navigation et l'édition de systèmes d'équations, et la mise en page multidimensionnelle de l'évaluation des expressions est faite. Le modèle du temps pour ces applications est complètement synchrone. Pour les systèmes synchrones distribués, trois modèles d'interaction sont proposés : pair-à-pair, travailleurs hiérarchiques et parent-éter.

Le chapitre 7, intitulé «La récursion d'en bas», explore l'utilisation et la réalisation de fonctions récursives, pour des fonctions simples, des filtres de flux de données, des algorithmes diviser-pour-régner et des algorithmes de programmation dynamique. Des techniques pour la génération de l'évaluation efficace de ces fonctions sont présentées.

Le chapitre 8, intitulé «Mettre le contrôle à l'index», considère l'inclusion d'effets de bord dans le langage TransLucid. L'idée clé est qu'un effet de bord est considéré comme une action qui ne peut avoir qu'une seule fois dans un contexte multidimensionnel donné, et que d'autres effets de bord, dans d'autres contextes, peuvent être obligés d'avoir lieu précédemment. La sémantique des programmes impératifs est présentée en TransLucid basée sur cette compréhension d'effets de bord. Le chapitre termine avec une discussion des paradigmes de programmation majeurs et des formes de conception orientée-object.

La conclusion continue l'exploration de l'espace cartésien, en focalisant sur des sujets pas couverts dans le reste dans le texte. Ceux-ci incluent les structures de données, les types de données et les algorithmes.

Partie I : Préparer l'espace cartésien

Chapitre 1 : De Lucid à TransLucid : L'itération, les programmations de flux de données, intensionnelle et cartésienne

Nous présentons le développement du langage Lucid à partir du Lucid originel des années 1970 au TransLucid d'aujourd'hui. Chaque version successive du langage a été une généralisation de langages précédents, mais permettant une compréhension accrue des problèmes étudiés.

Le Lucid originel (1976), conçu originellement pour des fins de vérification formelle, a été utilisé pour formaliser l'itération des programmes `while`. Le language pLucid (1982) a été utilisé pour décrire des réseaux de flux de données. Indexical Lucid (1987) a été introduit pour la programmation intensionnelle, dans laquelle la sémantique d'une variable fut comprise comme une fonction à partir d'un univers de mondes possibles vers les valeurs ordinaires. Avec TransLucid, et l'utilisation de contextes comme valeurs de première classe, la programmation peut être percue dans un cadre cartésien.

Chapitre 2 : Le versionnement par mondes possibles

Nous présentons une histoire de l'application de la sémantique des mondes possibles de la logique intensionnelle au développement de structures versionnées, variant de la simple configuration de logiciels à la mise en réseau d'applications distribuées sensibles au contexte pénétrées par de multiples contextes partagés.

Dans cette approche, toutes les composantes d'un système varient par rapport à un espace uniforme multidimensionnel de versions, ou de contextes, et l'étiquette d'une version construite est la plus petite borne supérieure des étiquettes de version des composantes choisies comme étant les plus pertinentes. Les deltas de contexte permettent la description de changements de contextes, de changements successifs des composantes et des systèmes d'un contexte vers un autre. Avec les éthers, des contextes actifs avec participants multiples, plusieurs programmes mis en réseau peuvent communiquer en diffusant des deltas à travers un contexte partagé auquel ils s'adaptent continuellement.

Partie II : Construire l'espace cartésien

Chapitre 3 : Une introduction à Core TransLucid

Nous introduisons ici le language Core TransLucid à travers une séries d'exemples simples. Dans la programmation cartésienne, comme pour le système des coordonnées cartésien, la clé est la multidimensionnalité. Pour les coordonnées, un point dans un espace unidimensionnel devient un ligne dans un espace bidimensionnel, un plan dans un espace tridimensionnel, un espace tridimensionnel dans un espace quadridimensionnel, et ainsi de suite. Similairement, dans la programmation cartésienne, toute entité est considérée comme «variante» dans toutes les dimensions, malgré le fait que cette «variance» peut être constante dans la plupart des dimensions.

La présentation dans ce chapitre débute avec une discussion du système de coordonnées cartésien, en focalisant sur la multidimensionnalité. Nous introduison le *tuple multidimensionnel*, qui est utilisé pour indexer des points, des lignes et d'autres structures dans cet espace. Nous continuons alors vers TransLucid et démontrons comment utiliser ce tuple et modifier des parties de ce tuple peuvent être utilisés pour naviguer à travers cet espace et provoquer le calcul.

Les primitives TransLucid sont présentées informellement à travers les exemples suivants : factoriel, Fibonacci, Ackermann ; coder des programmes de machines à registres illimités ; et matrices triangulaires. Ces exemples bien connus suffisent pour présenter le langage de base. Le reste de la présentation focalise sur deux types d'abstraction fonctionnelle—par valeur ou par nom—et sur l'utilisation de valeurs calculées comme dimensions.

Chapitre 4 : La syntaxe et sémantique de Core TransLucid

Dans ce chapitre, nous présentons formellement Core TransLucid. Nous débutons avec la syntaxe abstraite et la sémantique dénotationnelle, utilisant un environnement portant les identificateurs vers les *intensions*, qui portent les contextes vers les valeurs. Puis, nous introduisons une sémantique opérationnelle qui correspond structurellement à la sémantique dénotationnelle, en faisant des demandes pour des couples (*identificateur*, *contexte*), qui peuvent à leur tour provoquer des demandes pour d'autres couples similaires, et démontrons l'équivalence des deux sémantiques pour produire des valeurs correctes.

Les appels de fonction sont transformés en changements de contexte, par l'utilisation d'une dimension pour suivre les différentes occurrences d'application. Pour l'application par valeur, la dimension prend la valeur comme argument, tandis que pour l'application par nom, cette dimension a comme valeur une liste retenant l'ensemble des arguments (expressions) qui ont été passés dans des instantiations précédentes.

Nous démontrons dès lors comment la sémantique opérationnelle peut être adaptée afin de mémoriser des valeurs calculées précédemment, où la cache resemble l'environnement de la sémantique dénotationnelle, mais s'assure que seulement les dimensions pertinentes aux calculs sont mémorisées dans la cache.

Chapitre 5 : Construire un interprète

Dans ce chapitre, nous présentons l'interprète qui réalise le langage TransLucid [8]. La différence essentielle entre le langage Core TransLucid présenté dans les chapitres précédents et le langage dont la réalisation est décrite ici est la richessse et la diversité de types de données et opérations. En particulier, le TransLucid présenté ici est conçu comme un *langage de coordination*, ce qui signifie qu'il fournit une interface à d'autres structures écrites dans un langage hôte.

L'environnement de programmation TransLucid, disponible à `translucid.sourceforge.net`, est écrit en `C++0x`, la nouvelle norme pour `C++`. Pour la compilation, il nécessite actuellement GNU `g++ 4.5.0` (`gcc.gnu.org`) et les bibliothèques `C++` Boost `1.43.0` (`www.boost.org`).

En plus des trois types de données atomiques décrits dans le chapitre précédent, l'environnement soutient naturellement les types de données `intmp` (entier multi-précision GNU), `uchar` (caractère Unicode 32-bit) et `ustring` (chaîne Unicode 32-bit). De plus, les utilisateurs peuvent ajouter d'autres types de données et opérations en utilisant des bibliothèques, and ceux-ci peuvent être manipulés par TransLucid. De plus, le parseur très flexible peut être paramétrisé afin que de nouveaux opérateurs puissent être ajoutés, en forme préfixe, postfixe ou infixe, et les littéraux (constantes) de nouveaux types puissent être écrits aussi simplement que s'ils étaient des littéraux de types natifs.

L'interprète est conçu pour permettre l'interaction du langage hôte, ici `C++`, avec TransLucid : TransLucid peut appeler TransLucid ou `C++`, et `C++` peut appeler `C++` ou TransLucid. Le moyen par lequel cette communication a lieu est l'*hyperdaton*, un object qui répond au contexte (un *tuple*) et retourne une *valeur étiquetée*, un couple (*constante*, *tuple*), où le tuple encode le souscontexte utilisé pour calculer la constante.

L'hyperdaton est la structure de données clé. Un hyperdaton peut être généré à la main en `C++` ou bien peut être produit de manière automatique à partir d'équations TransLucid. Toutes les formes d'expression sont des sousclasses de la classe hyperdaton ; ceci est aussi le cas pour les variables, les équations et le système lui-même.

La *variable* est une spécialisation de l'hyperdaton, utilisée pour stocker toutes les informations à propos d'une variable, incluant ses équations définissantes, et éventuellement ses variables locales.

Le *système* est une spécialisation de variable, utilisée pour stocker un ensemble de variables, tout aussi avec un ensemble d'hyperdatons en entrée, utilisés par les équations définissant les variables, et un ensemble d'hyperdatons en sortie, remplis par l'utilisation de *demandes*, essentiellement des réservations pour des calculs qui doivent avoir lieu dans des contextes spécifiques.

La sémantique d'un système est synchrone. À chaque instant, les hyperdatons en entrée, les équations et les demandes doivent être mis à jour. Les demandes pertinentes sont alors calculées, en se faisant remplissant les hyperdatons en sortie. Le processus est répété à chaque instant.

Puisque le système cartésien doit être utilisé comme un système réel, nous devons être capables de gérer les types définis par les utilisateurs, les bibliothèques, les identificateurs de dimensions, et ainsi de suite. Ainsi, les parties internes d'un système sont conçues afin qu'elles puissent être comprises du point de vue TransLucid : ceci est fait en utilisant des dimensions et variables prédéfinies, afin que, par exemple, ajouter une nouvel opérateur dans un système revient effectivement à ajouter une nouvelle équation dans le système.

Partie III : Explorer l'espace cartésien

Chapitre 6 : Le temps pour les applications

Dans ce chapitre, nous présentons deux applications basées surTransLucid, `tlcore` et $\texttt{S}^3$, et examinons comme le temps est utilisé dans ces applications et dans des systèmes synchrones distribués.

L'application `tlcore` est un interprète textuel, avec une sémantique réactive synchrone. À chaque instant, on peut ajouter, remplacer ou supprimer des équations définissantes et des expressions à évaluer, et la dimension spéciale `time` peut être interrogée afin de prendre en compte l'évolution du système d'équations.

L'application $\texttt{S}^3$ est un navigateur graphique de code TransLucid et d'expressions, avec lequel on peut éditer un ensemble d'équations définissantes et d'expressions à évaluer, et on peut visualiser l'évaluation de ces expressions dans un espace multidimensionnel. À cause de la nature exploratoire de cet outil, le temps peut advancer par des micro-pas, chaque fois que les équations ou les expressions sont modifiées, ou par des macro-pas, quand les changements sont validés.

Ces applications ont permis le développement d'une sémantique temporisée du système TransLucid présentée dans le chapitre précédent. En utilisant cette sémantique, nous pouvons dès lors explorer différentes manières d'avoir de multiples systèmes TransLucid interagissantes. Nous présentons trois modèles : le modèle pair-à-pair, le modèle travailleur-hiérarchiques, et le modèle parent-éther.

Chapitre 7 : La récursion d'en bas

Dans ce chapitre, nous focalisons sur les réalisations efficaces de structures de données ou de fonctions définies récursivement. Nous considérons les fonctions récursives simples, les filtres de flux de données définis de manière récursive, et les algorithmes diviser pour régner et de programmation dynamique.

Chapitre 8 : Mettre le contrôle à l'index

Le language TransLucid est déclaratif. Une question est donc pertinente : La programmation cartésienne peut-elle être utilisée pour comprendre la complexité de la «programmation réelle», c-à-d avec des effets de bord et la programmation impérative ? Nous examinons ces thèmes dans ce chapitre, et démontrons que l'approche cartésienne aide à la clarification de certains problèmes.

Conclusions

La multidimensionnalité implicite dans la programmation cartésienne a été démontrée comme étant remarquablement versatile. Étant donné un problème, il existe un petit ensemble fixe de dimensions qui définissent le problème, avec un ensemble arbitrairement large de dimensions qui paramétrisent le dit problème.

L'ensemble des sujets qui suivent du document actuel est très grand. Au lieu de couvrir tous les sujets possibles, nous examinons en détail deux thèmes qui n'ont pas été discutés de manière substantive dans cette thèse : les structures de données et les types, et les algorithmes et la complexité.

Bien sûr, il existe d'autres sujets qui doivent aussi être examinés. Par exemple, il y a très peu de discussion d'analyse statique. Ce sujet très important peut être appliqué en plusieurs manières au développement de TransLucid, incluant la vérification de types, l'optimisation dans le temps et l'espace, et la vérification formelle. Étant donné que TransLucid est un langage complètement déclaratif, et que les variables ont déjà des déclarations multiples, nous pouvons ajouter encore des déclarations. Dans ce cas, pendant le processus de «bestfitting», si plusieurs déclarations s'avèrent être meilleures, alors toutes doivent être appliquées, et toute évaluation ou vérification devrait s'assurer que toutes les déclarations meilleures génèrent des résultats cohérents dans un contexte donné.

Similairement, il n'y pas de discussion de ce nous appelons la «programmation orientée-recherche» (malheureusement le terme est déjà consacré ailleurs), qui inclue toute forme de programmation dans laquelle une part considérable du travail consiste en chercher dans de grandes bases de données, telle la programmation logique, la programmation de bases de données et le «data mining». Actuellement, nous n'avons pas de sémantique pour ce genre de programmation, malgré le fait que l'utilisation de dimensions multiples est hautement pertinente.

En général, la réalisation d'un problème donné peut être compris comme nécessitant la paramétrisation de l'espace des solutions, basée sur les composantes disponibles pour le calcul, dans l'hiérarchie de la mémoire et pour la communication. La recherche selon ce thème devrait porter fruit, étant donné la grande variété de multiprocesseurs qui sortent actuellement sur le marché.

Quand Descartes a introduit son système de coordonnées, la présentation et la résolution d'un grand nombre de problèmes existants en géométrie a été simplifiée, et a permis la définition de problèmes beaucoup plus difficiles qui ne pouvaient pas être exprimés auparavant. Jusqu'à présent, chaque part de l'informatique que nous avons examiné sous la loupe cartésienne nous a permis de nouvelles découvertes. La programmation cartésienne nous permettra-t-elle aussi la spécification de problèmes jusqu'à présent inexprimables ?

Introduction

This thesis examines different aspects of computer science from a Cartesian perspective, attempting to provide a unified framework based on the Cartesian coordinate system. At the core of this work is the introduction of Cartesian Programming, in which all entities are assumed to vary with respect to a multidimensional context, and the TransLucid declarative programming language, implementing the ideas of Cartesian programming.

The key intuition is that even the simplest problems are multidimensional, and contain far more parameters than might initially be assumed. For example, if we consider the time that a falling body might take to reach the ground, we would likely begin with Newton's law, assuming that G is constant. But this problem is actually quite complicated, as there are myriads of relevant parameters: the exact value for G in the relevant location, the humidity of the air, current winds, the density of the object, the cross-section of the object, the magnetic properties both of the object and the location, and so on. And if the object flies, i.e., it is a bird or plane, everything changes.

For such problems, it is common to program assuming a limited set of parameters, then to add further parameters over time to take into account a wider set of situations or to achieve higher accuracy or control over the problem at hand. As this programming takes place, it will turn out that in a set of situations, one set of parameters will be preponderant, while in another set of situations, other parameters will be more important. In certain exceptional situations, all of the previous programming is placed aside, and completely new code needs to be written.

If we consider another branch of computing, such as rendering geographical information on a screen, then there are also numerous parameters: the geographical projection, the parameters for the specific projection chosen, the portion of the globe that is being presented, the screen resolution, the geographical resolution of the databases being selected, and so on. And even the text itself has numerous parameters: character encoding, script, language, use of transliteration, font, style, etc.

In Cartesian programming, *all* programmed entities are assumed to vary, right from the start, in *all possible dimensions*. Initially, one programs these entities through equations referring to a limited set of dimensions, and as time progresses, one adds further equations that are more specific in certain situations. The idea is that when a value is needed from a variable given a running context, the *bestfit* definitions for that variable, with respect to that context, are chosen. Then, if certain conditions are exceptional, these conditions can be specified in a new set of equations, and these equations will be chosen when needed. All this without changing the previous equations, unless these have been demonstrated to be erroneous, in which case they should be replaced.

Cartesian programming did not come to life through a great « Fiat Lux! » Therefore, the present work has as objective to present the key developments that led to the very conception of Cartesian programming; the design, semantics and implementation of the TransLucid language; and current applications that have been developed using TransLucid.

The main body of the thesis is divided into three parts. Part I situates the research on Cartesian programming in its historical context, outlining the key developments that preceded the coining of the term "Cartesian Programming". Part II presents the TransLucid language, its semantics, both denotational and operational, and the current implementation. Part III is used for exploration of several topics in computer science from the Cartesian perspective: a standalone TransLucid interpreter, a TransLucid code browser and hyperdation visualizer, the efficient implementation of recursive functions, control flow and side-effects, and programming paradigms and design patterns.

Part I, the history, consists of two physical chapters, but there are three interwoven threads: *intensional programming*, *synchronous programming* and *possible-worlds versioning*.

In 1984, when the author was still an undergraduate student at the University of Waterloo, he followed a course on the intensional logic of Richard Montague, which used possible-worlds semantics and Kripke structures to give the semantics for fragments of natural language. Three years later, while writing his PhD thesis at the *Institut National Polytechnique de Grenoble*, the author was surprised to come across a paper written by Anthony Faustini and William Wadge, entitled "Intensional Programming", which outlined how the Lucid language of the time could be understood using possible-worlds semantics.

Chapter 1 presents a history of the development of successive versions of the Lucid language from the original Lucid of 1975 through to the initial proposals of TransLucid. As the language evolved, so did the terms used to describe the forms of programming: dataflow programming, intensional programming, indexical programming and Cartesian programming. One can summarize the development as a process of “freeing the dimensions”. The very first Lucid paper used multidimensionality, but only one dimension could be manipulated at a time, the dimensions had a fixed order of appearance, and the dimensions were not directly accessible to the programmer. The TransLucid presented in this thesis considers all dimensions to be equal and allows all dimensions to be manipulated directly and simultaneously.

In synchronous programming, the key assumption is that inputs do not change while one is still calculating outputs; as a result, one may assume that the inputs and outputs are simultaneous. The LUSTRE language developed by Nicolas Halbwachs and Paul Caspi provided a synchronous semantics to a single-dimensional Lucid, with the intuition that the i-th entries of two streams on the same clock are generated within the same instant.

The history of synchronous programming has been written elsewhere [9]. The author did play a key rôle at the beginning, by writing for his PhD the first denotational and operational semantics and the first compiler for LUSTRE. This language is at the core of the successful Scade programming suite, currently marketed by Esterel Technologies, and used for programming the Airbus flight-control software, among other things.

Chapter 2 presents a history of the development of possible-worlds versioning. (Originally called *intensional versioning*, it was renamed to not clash with an existing use of that term.) Here, the possible-worlds semantics of intensional logic was applied to the *structure* of programs, as opposed to their behavior. Originally proposed for the purposes of version control of software, possible-worlds versioning was used to develop versioned forms of many different kinds of software: Web sites, object-oriented programming languages, Linux processes, etc.

As these different experiments were undertaken, new intuitions arose. The first was that the multidimensional context providing the indexing for the versioned entities could be given a *physical* interpretation: an entity is immersed in a *pervasive* context, so if the context is changed, then every aspect of the entity is informed of this change of context, and may adapt accordingly. The second intuition was that, given this physical context, more than one entity could share the same context, and these could communicate through broadcasting, simply by changing this shared context, which is now called an *æther*.

Cartesian programming incorporates both intensional programming and possible-worlds versioning. Both an entity’s structure and behavior, represented by the equations, will vary as the context varies. However, one dimension plays a primordial rôle: time. With time, everything in a system may evolve, even the set of dimensions available for the specification of variance. Given the author’s experience with synchronous programming, returning to Grenoble to present this habilitation thesis seems natural.

Part II presents the TransLucid language from the ground up. The three chapters contain the key technical details necessary for the proper development of the language.

Chapter 3 introduces the language with an intuitive presentation of a number of examples. The chapter is standalone, and can be understood with a basic knowledge of high-school mathematics. It begins with a recapitulation of the Cartesian coordinate system and can be read without having to delve into semantic details.

When an expression in a Cartesian program is executed, there is always a running context, a mapping from a (possibly infinite) arbitrary set of dimensions to values. During the evaluation of an expression, the context can be perturbed by changing the values of some of the dimensions or can be queried, dimension by dimension. When a variable is encountered, there may be a number of definitions that are applicable in the current context. The *bestfit* definitions are chosen, necessarily according to a finite set of the available dimensions, and evaluation proceeds.

Functions can take two different kinds of parameters: call-by-value parameters, used to pass dimensions and constants, are evaluated prior to entry in a function body; call-by-name parameters, used to pass hyperdatons, are evaluated on demand within the function body.

Chapter 4 presents the semantics for TransLucid, in three parts. First is the denotational semantics, which is given through fixed-point semantics over arbitrary dimensional structures defined through sets of recursive equations, i.e., with bottom-up evaluation. Second is a demand-driven operational semantics, which only evaluates those expressions that need to be evaluated. Third is a multi-threaded demand-driven operational semantics for TransLucid, using a context-dependent cache. The evaluator and the cache play a game, ensuring that entries in the cache only refer to dimensions actually encountered while evaluating expressions.

Chapter 5 transforms the Core TransLucid language into a real programming environment, usable as a coordination language on top of `C++`. A TransLucid system is a reactive system, which at each instant, has a current set of equations, a current set of inputs, outputs and demands. When the evaluation for an instant begins, all demands that are applicable are evaluated, with their results placed in the appropriate outputs.

The key data structure is the *hyperdaton*, a `C++` functor which returns a value when passed a context as input. All parsed constants, variables, expressions and equations are subclasses of the hyperdaton class. This class can also be subclassed to provide an interface permitting the creation of objects usable both in `C++` and in TransLucid.

Facilities are also provided for the use of user-defined data types and operations, both at the semantic and the syntactic level, and the highly flexible parser allows the introduction of new prefix, postfix and infix operators.

Part III is entitled “Explorations in Cartesian Space”.

Chapter 6, entitled “Time for Applications”, presents two applications that use the TransLucid library implementation. The first is a standalone interactive loop, in which the system of equations is actually a synchronous reactive system: at each instant, equations may be added, deleted or replaced, only affecting the future behavior of the system. The second is a graphical system allowing the browsing and editing of systems of equations, and the multidimensional display of expressions being evaluated. The model of time for these standalone applications is completely synchronous. For distributed synchronous systems, three models of interaction are suggested: peer-to-peer, hierarchical-worker and parent-æther.

Chapter 7, entitled “Bottom-up Recursion”, explores the use and implementation of recursive functions, for simple functions, dataflow filters, divide-and-conquer algorithms and dynamic programming algorithms. Techniques for the generation of efficient evaluation of these functions are presented.

Chapter 8, entitled “Putting Control on the Index”, considers the inclusion of side-effects in the TransLucid language. The key idea is that a side-effect is considered to be an action that may only be undertaken once in a given multidimensional context, and that other side-effects, in other contexts, may be forced to be undertaken previously. The semantics of imperative programs is presented in TransLucid based on this understanding of side-effects. The chapter finishes with discussion of the major programming paradigms and object-oriented design patterns.

The conclusion continues the exploration of Cartesian space, by focusing on topics not covered within the rest of the text. These include data structures, data types and algorithms.

Part I

Preparing the Cartesian Space

Chapter 1

From Lucid to TransLucid: Iteration, Dataflow, Intensional and Cartesian Programming

(with Blanca Mancilla and Gabriel Ditu)
Mathematics in Computer Science 2(1):37–61, 2008
`http://dx.doi.org/10.1007/s11786-008-0043-9`

We present the development of the Lucid language from the Original Lucid of the mid-1970s to the TransLucid of today. Each successive version of the language has been a generalization of previous languages, but with a further understanding of the problems at hand.

The Original Lucid (1976), originally designed for purposes of formal verification, was used to formalize the iteration in `while`-loop programs. The pLucid language (1982) was used to describe dataflow networks. Indexical Lucid (1987) was introduced for intensional programming, in which the semantics of a variable was understood as a function from a universe of possible worlds to ordinary values. With TransLucid, and the use of contexts as first-class values, programming can be understood in a Cartesian framework.

1.1 Introduction

This paper presents the development of the Lucid programming language, from 1974 to the present, with a particular focus on the seminal ideas of William (Bill) Wadge. These include the use of infinite data structures, the importance of iteration, the use of multidimensionality, the rise of intensional programming, the importance of demand-driven computation, eduction as a computational model, and the necessity of replacing the von Neumann architecture with more evolved computational machines.

Many of these ideas were explicit right from the beginning, others implicit, while still others were developed through a series of implementations and expansions of Lucid. Finally, some had to wait until the design and implementation of the most recent version, TransLucid, the result of many years of research.

The relevance of Wadge's ideas is increasingly timely. Let us consider the very last topic, with respect to computer architecture. Since 2003, single processor speedup has not kept pace with Moore's Law. The law remains valid, with chip transistor density doubling approximately every 24 months [63, 62]. However, a corresponding annual 52% single processor speedup, starting in 1986, ceased to be true in 2003, dropping to 20% [1]. To compensate, vendors have moved towards multicore processors, and researchers are talking about manycore processors, each capable of managing very large numbers of threads.

The problem with these hardware developments is that the mainstream programming languages are not well suited to these new architectures. It is difficult to transform a program written in `C` or some other imperative language to take advantage of parallelism available in a new architecture, let alone to take advantage of varying amounts and forms of parallelism, as successive architectures are brought onto the market every few months.

This kind of scenario was predicted by Wadge and Ed Ashcroft, in the introduction to their 1985 book, *Lucid, the Dataflow Programming Language* [95]. In the introduction, they spoofed the different researchers working in programming language design, semantics and implementation, categorizing them into Cowboys, Boffins, Wizards, Preachers and Handymen, according to their various preferences. The point of this tongue-in-cheek description was not meant to insult anyone — although a few feathers did get rustled — but, rather, to point out that most of these approaches implicitly assumed that the von Neumann architecture was going to remain with us forever, and that "Researchers who try to avoid the fundamental controversies in their subject risk seeing their lifetime's work prove in the end to be irrelevant."

The key insight of Bill Wadge is that significant advances in programming language design and semantics cannot be made independently of the underlying computational models. Efficiency is not a mere implementation detail, allowing a programmer to simply provide some unexecutable specification. As a result, existing programming practices, although possibly limited, cannot be ignored. Crucially, the most important practice is that computers iterate, i.e., they are good at doing things over and over again, and do not recurse.

Focusing on efficiency, one must be careful to analyze the underlying assumptions that are being made in any given computational model. For example, one of the criticisms made towards Lucid and its implementations is that the demand-driven implementations are inefficient. Although it is true that demand-driven mechanisms do carry an overhead, it is rarely acknowledged that *any* computational model using memory is itself demand-driven. The infamous Von Neumann memory bottleneck is a bottleneck *precisely* because the memory is accessed in a demand-driven manner: one gives the index of a cell and makes a demand for the value therein; depending on the structure of the memory hierarchy, this demand will be treated in different ways.

The point, therefore, is not whether one should use demand-driven or data-driven mechanisms but, rather, exactly what kind of demand-driven mechanisms are most suitable? Or, given appropriate architectures, to what extent can demand-driven mechanisms be translated into data-driven systems? The first question is completely compatible with the current trends in computer architecture, with multiple cores each running multiple threads; if a demand in one thread blocks, it may well be the case that a previously blocked demand in another thread has been resolved. The second question deals with the development of innovative architectures.

In all of the variants of Lucid, infinite data structures are defined using mutually recursive systems of equations. The recursion is uniquely for definitional purposes, it is not a computational phenomenon. One iterates towards a result.

In this article, we examine the successive versions of Lucid and examine, through the use of common examples, the different interpretations and ideas associated with these different versions. The general tendency is to move from sequential forms of computing to indexical forms of computing, leading ultimately to Cartesian programming with TransLucid.

1.2 Iteration: Original Lucid

The Lucid language was first conceived in 1974 by Bill Wadge and Ed Ashcroft when the two were academics at the University of Waterloo. Two major papers were published, one in 1976 in *SIAM Journal of Computing* [2], the other in 1977 in *Communications of the ACM* [3]. As we shall see below, in the (Original) Lucid they presented in these papers, Wadge and Ashcroft introduced infinite data structures, iteration and multidimensionality as means to formally describe computation.

At the time, discussions around structured programming were standard. In 1968, Edsger Dijkstra had penned his famous "Go to Statement Considered Harmful" article [26], making the computer science community realize that programming was not simply something that had to be done but, rather, something that could be done with elegance and grace. One of the main ideas, associated with Tony Hoare, was that a block should have a single entry point and a single exit point. These ideas were well presented in the books *Structured Programming* by Dahl, Dijkstra and Hoare [23], and *A Discipline of Programming* by Dijkstra [27]. In the latter, Dijkstra's presentation led naturally to the vision that computer programs could be formally verified if they had a proper mathematical description.

It is in this context that Wadge and Ashcroft began the work leading to Lucid [92]. Wadge's PhD work at Berkeley was in descriptive set theory, leading to the development of *Wadge games*, described in [84]. Given this experience, he was habituated in thinking in terms of infinite sets.

Wadge was examining programs such as the following one:

```
$I = 0$
$J = 0$
while (...)
    $J = J + 2 * I + 1$
    $I = I + 1$
    PRINT $J$
end while
```

which gives the output:

$$1\ 4\ 9\ 16\ \cdots$$

In this program, it is easy to understand and to prove that after the assignment:

$$J = J + 2 * I + 1$$

that $J = (I+1)^2$ and that after the assignment:

$$I = I + 1$$

that $J = I^2$. However, in programs of the form:

```
while (...)
    $J = \cdots$
    $P = \cdots$
    $J = \cdots$
    $P = \cdots$
end while
```

it is much more difficult to understand the meaning of a program, because of the reassignments of J and P. This study led to Wadge's insight of "(Re)Assignment Considered Harmful". By letting variables define infinite sequences, he could rewrite the above program as:

```
first $I = 0$;
 next $I = I + 1$;
first $J = 0$;
 next $J = J + 2 * I + 1$;
```

Hence:

$$\begin{aligned} I &= \langle 0, 1, 2, 3, \ldots \rangle \\ \texttt{next}\ I &= \langle 1, 2, 3, 4, \ldots \rangle \\ J &= \langle 0, 1, 4, 9, \ldots \rangle \\ \texttt{next}\ J &= \langle 1, 4, 9, 16, \ldots \rangle \end{aligned}$$

By introducing the operators `first` and `next`, one could define the entire history of a variable using just two lines. Formally, if the variables X and Y are defined by:

$$X = \langle x_0, x_1, x_2, \ldots, x_i, \ldots\rangle \tag{1.1}$$
$$Y = \langle y_0, y_1, y_2, \ldots, y_i, \ldots\rangle \tag{1.2}$$

then:

$$\texttt{first}\ X = \langle x_0, x_0, \ldots, x_0, \ldots\rangle \tag{1.3}$$
$$\texttt{next}\ X = \langle x_1, x_2, \ldots, x_{i+1}, \ldots\rangle \tag{1.4}$$

When a constant `c` appears in a program, it corresponds to the infinite sequence:

$$\texttt{c} = \langle c, c, c, \ldots, c, \ldots\rangle \tag{1.5}$$

Finally, when a data operator `op` appears in a program, it is applied pointwise to its arguments:

$$X\ \texttt{op}\ Y = \langle x_0 \text{ op } y_0,\ x_1 \text{ op } y_1,\ \ldots,\ x_i \text{ op } y_i,\ \ldots\rangle \tag{1.6}$$

Finally, to end an iteration, the `asa` operator is used:

$$X\ \texttt{asa}\ Y = \langle x_j, x_j, x_j, \ldots, x_j, \ldots\rangle, \qquad y_j \wedge \forall(i < j)\,\neg\, y_i \tag{1.7}$$

Here, it is assumed that the values of sequence Y must be convertible to Boolean values.

As can be seen from the above discussion, Wadge and Ashcroft privileged *iteration.* They did not redefine what a computer was doing but, rather, presented a formal framework in which one could state exactly what a computer was doing. The language they had introduced had a perfectly clear mathematical semantics, yet represented programming as it really existed.

To handle nested loops, the "time" index was extended to include "multidimensional time". A variable F of n dimensions, instead of being a mapping from $\mathbb{N}$ to values, becomes a mapping from $\mathbb{N}^n$ to values. The notation

$$F_{t_1\,t_2\,\cdots\,t_n}$$

denotes the element where the outermost time dimension has value t_n, and the innermost time dimension has value t_1.

The `latest` operator was used to freeze the value of the current stream representing the outer loop, while reaching into the inner loop to manipulate the relevant stream, so that one could come back to the outer loop with the result. Here are the definitions.

$$(\texttt{first}\ F)_{t_1\,t_2\,\cdots\,t_n} = F_{0\,t_2\,\cdots\,t_n} \tag{1.8}$$
$$(\texttt{next}\ F)_{t_1\,t_2\,\cdots\,t_n} = F_{(t_1+1)\,t_2\,\cdots\,t_n} \tag{1.9}$$
$$(F\ \texttt{asa}\ G)_{t_1\,t_2\,\cdots\,t_n} = F_{j\,t_2\,\cdots\,t_n}, \qquad G_{j\,t_2\,\cdots\,t_n} \wedge \forall(i < j)\,\neg G_{i\,t_2\,\cdots\,t_n} \tag{1.10}$$
$$(\texttt{latest}\ F)_{t_0\,t_1\,t_2\,\cdots\,t_n} = F_{t_1 t_2 \cdots\,t_n} \tag{1.11}$$

Below is an example of the use of two time dimensions, with the `latest` operator being used to access the values from within the nested loop. The program determines if n, the first entry in the *input*, is prime:

```
$n = \texttt{first}\ input$;
$\texttt{first}\ i = 2$;
    $\texttt{first}\ j = \texttt{latest}\ i * \texttt{latest}\ i$;
    $\texttt{next}\ j = j + \texttt{latest}\ i$;
    $\texttt{latest}\ idivn = (j \equiv \texttt{latest}\ n)\ \texttt{asa}\ (j \geqslant \texttt{latest}\ n)$;
$\texttt{next}\ i = i + 1$;
$output = \neg idivn\ \texttt{asa}\ (idivn \vee i * i \geqslant n)$;
```

So we have that:

$$
\begin{aligned}
i &= \langle 2, 3, 4, 5, \ldots \rangle \\
j &= \big\langle \langle 4, 6, 8, 10, \ldots \rangle,\ \langle 9, 12, 15, 18, \ldots \rangle,\ \ldots,\ \langle i_k^2, i_k^2 + i_k, i_k^2 + 2i_k, \ldots \rangle,\ \ldots \big\rangle
\end{aligned}
$$

and so on.

When Bill Wadge moved to the University of Warwick in the UK, he met David May, who suggested that the Lucid streams could be understood using dataflow networks, and that the `first` and `next` operators could be combined using a binary operator called `fby` ("followed by"):

$$
(F \ \texttt{fby}\ G)_{t_0\, t_1\, t_2 \cdots} = \begin{cases} F_{0\, t_1\, t_2 \cdots}, & t_0 = 0 \\ G_{(t_0 - 1)\, t_1\, t_2 \cdots}, & t_0 > 0 \end{cases} \tag{1.12}
$$

The above example then becomes:

```
n = first input;
i = 2 fby i + 1;
    j = (latest i * latest i) fby (j + latest i);
    latest idivn = (j ≡ latest n) asa (j ⩾ latest n);
next i = i + 1;
output = ¬idivn asa (idivn ∨ i * i ⩾ n);
```

With the repeated use of `latest`, Original Lucid variables could in theory become infinite entities of arbitrary dimensionality. Although much of the discussion around the early versions of Lucid, both here and elsewhere, has focused on Lucid variables as sequences, effectively privileging an implicit dimension called "time", Lucid variables have always been multidimensional. However, with the initial set of primitives, only one dimension could be manipulated at a time, and the elements of a sequence were seen to be evaluated in order.

The "Extensions" section of [6] included a clear research agenda on user-defined functions, recursive functions, non-pointwise functions, the `whenever` operator, and so on. It also drew an analogy between Lucid's assumed dataflow execution model and the dataflow networks of Kahn and MacQueen [48, 49], both at the semantic and the operational levels. As we can see, the initial Lucid papers contained far more than was apparent on the surface.

1.3 Lucid, the Dataflow Programming Language

The pLucid language is the version of Lucid presented in *Lucid, the Dataflow Programming Language*, by Wadge and Ashcroft [95]. In pLucid, the implicit nesting of iteration by indentation is replaced by the use of `where` clauses, corresponding to the `whererec` of ISWIM [53]. In addition, new operators and extra syntax for structuring programs are added.

Unlike in the original ISWIM, pLucid is a first-order language, disallowing higher-order functions. pLucid became the "Dataflow Programming Language" because it operates on infinite streams, and dataflow streams could now be naturally expressed. For the authors, these sequences were meant to be "histories of dynamic activity".

With pLucid's focus on dataflow, it became possible to recursively define filters over streams, using the primitive operators `first`, `next` and `fby`. We give below the definitions for the predefined operators `wvr`, `upon` and `asa`.

The `wvr` operator accepts two streams and outputs the value of the first stream whenever the second one is true. In other words, certain values of the first stream are suppressed, depending on the values of the second stream:

```
X wvr Y = if first Y
          then X fby next X wvr next Y
          else next X wvr next Y
          fi;
```

Hence if:

$$Y = \langle \texttt{false}, \texttt{false}, \texttt{true}, \texttt{true}, \texttt{false}, \texttt{true}, \ldots \rangle \tag{1.13}$$

then:

$$X \texttt{ wvr } Y = \langle x_2, x_3, x_5, \ldots \rangle \tag{1.14}$$

The `upon` operator accepts two streams and continually outputs the first value of the first stream until the second stream is next true. The next value of the first stream is then accepted and the process begins all over again. In other words, the first stream is *advanced upon* the second stream taking true values:

```
X upon Y = X fby if first Y
                 then next X upon next Y
                 else X upon next Y
                 fi;
```

Hence, if Y is as defined in Equation (1.13) then:

$$X \texttt{ upon } Y = \langle x_0, x_0, x_0, x_1, x_2, x_2, x_3, \ldots \rangle \tag{1.15}$$

The `asa` operator, described in §1.2, returns the value of the first stream that corresponds to the first true value in the second stream. In other words, `asa` is capable of selecting a single value from a stream. It can be used for simulating the halting clause of a loop or for terminating a program once it has computed the desired result.

$$X \texttt{ asa } Y = \texttt{first } (X \texttt{ wvr } Y)$$

Hence, if Y is as defined in Equation (1.13) then:

$$X \texttt{ asa } Y = \langle x_2, x_2, x_2, \ldots \rangle \tag{1.16}$$

Like in the Original Lucid, programs in pLucid work with a sort of multidimensional time, but the dimensions are not explicit. However, the `latest` operator is replaced with the `is current` operator to freeze values for iteration in the next dimension. The example from the previous section becomes:

```
¬idivn asa idivn ∨ i ∗ i ⩾ N
where
  N is current n;
  i = 2 fby i + 1;
  idivn = j ≡ N asa j ⩾ N
  where
    I is current i;
    j = I ∗ I fby j + I;
  end
end
```

Wadge and Ashcroft do note that a similar effect to using `is current` could be achieved by adding extra dimensions [95, p. 106].

With the recursive definitions, it becomes possible to use `next` so that the order of calculations does not correspond to the stream order. For example, given the definition:

```
howfar
where
  howfar = if X ≡ 0 then 0 else 1 + next howfar;
end
```

and variable X:

$$X = \langle 1, 6, 4, 0, 9, 2, 1, 4, 3, 0, 6, \ldots \rangle$$

then here is *howfar*:

$$\mathit{howfar} = \langle 3, 2, 1, 0, 5, 4, 3, 2, 1, 0, \ldots \rangle$$

We will see in the next section how the implementation moves from simple iteration to *eduction*.

1.4 Implementing Lucid: Eduction

Interpreters were written for the Original Lucid by Tom Cargill and David May. Despite the intuition that they were dealing with dataflow streams, they in fact implemented a demand-driven approach. A request is made for a (*variable*, *tag*) pair. Should the evaluation of this variable at that tag require the evaluation of other (*variable*, *tag*) pairs, then requests are made for these too. This technique, known as *eduction*,[1] has become standard in interpreters of all versions of Lucid.

The use of eduction is crucial to deal with two major problems. First, out-of-order computation may be needed, to access streams at any point. Second, some of the operators acting like filters need to discard some of their input. With eduction, there are no wasted calculations.

The original interpreters were replaced by a more refined version developed by Calvin Ostrum [65]. However one problem still remained, and this was that the interpreter suffered from poor performance, since certain values continually needed to be recalculated. The solution was the implementation of a *warehouse* to cache calculated values and accelerate future computations.

The housekeeping of the warehouse posed its own problems, as each tagged value had to be unique. And because of tagging, the size of the warehouse could increase quite rapidly. Tony Faustini extended Ostrum's interpreter and implemented a warehouse with a garbage collector [95] as follows: values in the warehouse have a retirement age; they grow older after successive garbage collections and they are deleted if not accessed by the time they reach their retirement age. The retirement threshold is determined dynamically as the warehouse is used. Although efficiency is not guaranteed, the scheme has been proven in practice. It is important to note that the performance of the warehouse affects the runtime performance of the program but not its correctness.

From this implementation it was clear that there was a tension between the denotational semantics of infinite sequences and the operational semantics of iteration: the objects being implemented were not pipeline data, as described in the semantics. Rather, they were objects being accessed randomly. Data could be added to an object for any given tag, and any given tag could be reached at any time. The language could also handle several of these objects at the same time. Basically the semantics and the implementation did not coincide. And they did not do so, because they could not talk of multidimensionality, even though many of the tools had already been developed.

Two choices were possible. First was to restrict Lucid so that it could become a pure dataflow language; this choice was taken with the design of LUSTRE, described in §1.5. Second was to find a better model for understanding Lucid; this choice was taken with the introduction of *intensional programming*, described in §1.7.

In fact, the two choices can be distinguished from a denotational point of view. In Lucid, for a one-dimensional stream, the order on streams is the *Scott* order, in which, for example:

$$\langle \bot, 1, \bot, 3, \bot, \bot, \bot, \ldots \rangle \quad \sqsubseteq_S \quad \langle \bot, 1, \bot, 3, \bot, 5, \bot, \ldots \rangle$$

Ordinary dataflow, on the other hand, uses the *prefix* order, for example:

$$\langle 0, 1, 2, 3 \rangle \quad \sqsubseteq \quad \langle 0, 1, 2, 3, 4, 5 \rangle$$

[1]To *educe* means to *draw out*.

1.5 Synchronous programming: LUSTRE

LUSTRE (Synchronous Real-Time Lucid) [15, 42] is a simplification of the Lucid language specifically designed for the programming of reactive systems in automatic control. LUSTRE was designed by Paul Caspi and Nicolas Halbwachs, and the first compiler and semantics were written by author Plaice.

LUSTRE uses the *synchronous* approach to programming such systems, in which it is assumed that the reaction to an input event always takes less time than the minimum delay between two successive input events. As a result, it can be assumed that the output generated from an input event is simultaneous to that input event.

A LUSTRE stream is a pipeline dataflow, using the prefix order over streams. It is assumed that all elements of a stream will be generated, in the same order as their indices. Different streams may have different clocks, either a global base clock or a Boolean LUSTRE dataflow.

The two main operators of LUSTRE are `->` and `pre`. The equation:

$$X = 0 \texttt{ -> } \texttt{pre}\ X + 1$$

is in some sense equivalent to the Lucid equation:

$$X = 0 \texttt{ fby } X + 1$$

However, the Lucid equation has no sense of timing, while the LUSTRE equation is timed. Furthermore, the `pre` corresponds to a delay in automatic control. With the LUSTRE primitives, it is impossible to refer to the future, nor to more than a finite amount of the past.

When several dataflows share the same clock, then the i-th element of each stream sharing that clock must be calculated within the i-th instant of that clock. This approach, a radical simplification of Lucid, is very powerful for representing timed systems.

Today, LUSTRE is the core language in the Scade Toolkit, distributed by Esterel Technologies, and used for programming control systems in nuclear reactors, avionics and aerospace systems [34]. At the time of writing, Scade is used for programming part of the flight control or the engine control of the following aircraft:

- Airbus A380, A340-500, A340-600 (EU).
- Sukhoi SuperJet 100 (Russia).
- Eurocopter Écureuil/Astar AS 350 B3 (EU).
- Embraer Phenom 100 (Brazil), with Pratt-Whitney PW617F.
- Cessna Citation Encore+ (USA), with Pratt-Whitney PW535B.
- Dassault Aviation Falcon 7X (France).

1.6 Explicit multidimensionality: Ferd Lucid and ILucid

The "Beyond Lucid" chapter in [95] presents a discussion of explicitly making Lucid multidimensional, in order to not have to use operators such as `is current`. Two approaches are studied.

Ferd Lucid Ferd Lucid [95, pp. 217–22] was defined so that one could manipulate infinite arrays varying in one time dimension and an arbitrary number of space dimensions. These arrays were called *ferds* (an obsolete word in the Oxford English Dictionary meaning "warlike arrays"). The notation

$$F_t^{s_0\, s_1\, \cdots}$$

denotes the element where the time dimension has value t, the first space dimension has value s_0, the second space dimension has value s_1, etc. Here are the operators:

$$(\texttt{first}\ F)_t^{s_0\,s_1\,s_2\,\cdots} = F_0^{s_0\,s_1\,s_2\,\cdots} \tag{1.17}$$

$$(\texttt{next}\ F)_t^{s_0\,s_1\,s_2\,\cdots} = F_{t+1}^{s_0\,s_1\,s_2\,\cdots} \tag{1.18}$$

$$(F\ \texttt{fby}\ G)_t^{s_0\,s_1\,s_2\,\cdots} = \begin{cases} F_0^{s_0\,s_1\,s_2\,\cdots}, & t = 0 \\ G_{t-1}^{s_0\,s_1\,s_2\,\cdots}, & t > 0 \end{cases} \tag{1.19}$$

$$(\texttt{initial}\ F)_t^{s_0\,s_1\,s_2\,\cdots} = F_t^{0\,s_0\,s_1\,s_2\,\cdots} \tag{1.20}$$

$$(\texttt{rest}\ F)_t^{s_0\,s_1\,s_2\,\cdots} = F_t^{(s_0+1)\,s_1\,s_2\,\cdots} \tag{1.21}$$

$$(F\ \texttt{cby}\ G)_t^{s_0\,s_1\,s_2\,\cdots} = \begin{cases} F_t^{s_1\,s_2\,s_3\,\cdots}, & s_0 = 0 \\ G_t^{(s_0-1)\,s_1\,s_2\,\cdots}, & s_0 > 0 \end{cases} \tag{1.22}$$

The operators `initial`, `rest` and `cby` are counterparts to the Lucid operators `first`, `next` and `fby`. However, they do not have identical semantics. As can be seen above, the *rank* — or *dimensionality* — of X `cby` Y is one more than the rank of Y, while the rank of `rest` X is one less than the rank of X. This approach is cumbersome, as the programmer needs to keep track of the rank of each of the objects being manipulated.

There are also operators to transform a space dimension into a time dimension, and vice versa:

$$(\texttt{all}\ F)_t^{s_0\,s_1\,s_2\,\cdots} = F_{s_0}^{s_1\,s_2\,\cdots} \tag{1.23}$$

$$(\texttt{elt}\ F)_t^{s_0\,s_1\,s_2\,\cdots} = F_t^{t\,s_0\,s_1\,s_2\,\cdots} \tag{1.24}$$

We give an example to compute the prime numbers using the sieve of Erasthones:

```
all p
where
  i = 2 fby i + 1;
  m = all i fby m wvr m mod p ≠ 0;
  p = initial m;
end
```

The behavior of this program is as follows.

$$i = \langle 2, 3, 4, 5, 6, \ldots \rangle$$

$m =$

$s_0 \backslash t$	0	1	2	3	4	$\cdots$
0	2	3	5	7	11	$\cdots$
1	3	5	7	11	13	$\cdots$
2	4	7	11	13	17	$\cdots$
3	5	9	13	17	19	$\cdots$
4	6	11	15	19	23	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

$$p = \langle 2, 3, 5, 7, 11, \ldots \rangle$$

The result is that of p, but varying in space dimension s_0.

The main drawback of Ferd Lucid is that the extensional treatment of arrays (i.e., a 3D object is a sequence of 2D objects and a 2D object is a sequence of 1D objects) makes the language too difficult to use in solving multidimensional problems, where the dimensions are not only orthogonal but transposable.

ILucid ILucid [95, pp. 223–7] was defined to manipulate "multidimensional time." The `active` operator decreases the rank of its argument, while `contemp` increases its rank. The notation

$$F_{t_0\,t_1\,t_2\,\cdots}$$

denotes the element where the first time dimension has value t_0, the second time dimension has value t_1, the third time dimension has value t_2, etc. Here are the operators:

$$(\texttt{first}\ F)_{t_0\,t_1\,t_2\cdots} = F_{0\,t_1\,t_2\cdots} \tag{1.25}$$
$$(\texttt{next}\ F)_{t_0\,t_1\,t_2\cdots} = F_{(t_0+1)\,t_1\,t_2\cdots} \tag{1.26}$$
$$(F\ \texttt{fby}\ G)_{t_0\,t_1\,t_2\cdots} = \begin{cases} F_{0\,t_1\,t_2\cdots}, & t_0 = 0 \\ G_{(t_0-1)\,t_1\,t_2\cdots}, & t_0 > 0 \end{cases} \tag{1.27}$$
$$(\texttt{active}\ F)_{t_0\,t_1\,t_2\cdots} = F_{t_0\,t_2\,t_3\cdots} \tag{1.28}$$
$$(\texttt{contemp}\ F)_{t_0\,t_1\,t_2\cdots} = F_{t_0\,t_0\,t_1\,t_2\cdots} \tag{1.29}$$

A number of "interesting" operators could be defined. For example:

$$(\texttt{current}\ F)_{t_0\,t_1\,t_2\cdots} = F_{t_1\,t_1\,t_2\cdots} \tag{1.30}$$
$$(\texttt{remaining}\ F)_{t_0\,t_1\,t_2\cdots} = F_{(t_1+t_0)\,t_1\,t_2\cdots} \tag{1.31}$$

Suppose that:

$$X = \langle\langle a_0, a_1, a_2, \ldots\rangle, \langle b_0, b_1, b_2, \ldots\rangle, \langle c_0, c_1, c_2, \ldots\rangle, \ldots\rangle$$

then:

$$\begin{aligned} \texttt{current}\ X &= \langle\langle a_0, a_0, a_0, \ldots\rangle, \langle b_1, b_1, b_1, \ldots\rangle, \langle c_2, c_2, c_2, \ldots\rangle, \ldots\rangle \\ \texttt{remaining}\ X &= \langle\langle a_0, a_1, a_2, \ldots\rangle, \langle b_1, b_2, b_3, \ldots\rangle, \langle c_2, c_3, c_4, \ldots\rangle, \ldots\rangle \end{aligned}$$

For example, the following program:

```
contemp (y asa y > n)
where
   n = current active m;
   y = remaining active x;
end
```

finds the first-encountered present or future value of x which is greater than the present (current) value of m.

However, the set of "interesting" operators is unbounded, so more general mechanisms were needed.

1.7 Intensional programming: Field Lucid

The aforementioned chasm between denotational and operational issues was recognized by the authors of Lucid. Ashcroft wrote [4]:

> There was no unifying concept that made sense of it all, apart from the mathematical semantics. It was difficult to explain the language operationally. We used statements like "the values of variables are infinite sequences, but don't think of them as infinite sequences — think of them as changing."

This conundrum was solved in a 1986 paper by Faustini and Wadge entitled "Intensional Programming" [35]. In this paper, they made an explicit analogy between Lucid programs and the intensional logic of Richard Montague [91, 30].

The objective of Montague's work was to give a formal semantics to a significant subset of natural language, basing himself on prior work by Rudolph Carnap [14] and Saul Kripke [50]. Carnap had already made the distinction between the *extension* of an utterance — the specific meaning in the exact context of utterance (point in time and space, speaker, listener, etc.) — and its *intension* — the overall meaning for all of the possible contexts of utterance. Kripke had developed a means for using *possible worlds* as indices for giving the semantics of modal logic. Montague's work was to create a new logic, with a rich set of modal operators, using *Kripke structures*.

While developing a semantics for the warehouse, Faustini and Wadge discovered Montague's work, and understood that it is directly applicable to the variables of Lucid. Expressions become intensions mapping possible worlds (multidimensional tags) to extensions (ordinary values). Lucid's operators can be understood as intensional context-switching operators that manipulate the time dimension: `next` moves forward one timepoint and `fby` moves back one timepoint. Simple operations, such as addition, previously treated as pointwise operations on infinite sequences, were simply applied to single values under particular contexts.

The possible worlds semantics of intensional logic significantly clarifies the use of dimensions in Lucid. The dimensions define a coordinate system and each point in multidimensional space is a separate possible world. Lucid's "time" dimension is no longer a conceptual prop. In the words of Ashcroft, "Intensionality clears up the confusion."

Consider Lucid programs using just `next` and `fby`. Then there exists a set of possible worlds, indexed by the natural numbers, $\mathbb{N}$. When the i-th value of stream X is being requested, the understanding should be that we are *in* world i and we are simply asking for the value of X (in that world). The current possible world can be determined using the `#` primitive, and the `@` primitive can be used to access values in other possible worlds. Both `#` and `@` are *intensional operators*, as are derived operators such as `fby` and `next`. A variable X of type $\mathbb{D}$ defines an intension, a mapping $\mathbb{N} \to \mathbb{D}$. The value of X in a single possible world is an *extension*.

Once Lucid was understood as an intensional language, further developments consisted of creating more complex, multidimensional, universes of possible worlds, along with the appropriate syntactic adjustments. The first version of Lucid taking this approach was Field Lucid. Its multidimensional data structures can vary independently in multiple orthogonal dimensions. The Lucid operators are expanded to an arbitrary number of dimensions: for every $i \geqslant 0$, the operators $\texttt{initial}_i$, $\texttt{succ}_i$, $\texttt{sby}_i$, are equivalent to `first`, `next`, and `fby` respectively, but applied to the specific dimension as specified by the suffix. Multidimensional objects can be manipulated, but the dimensions cannot be manipulated, exchanged, transposed or even less created on the fly or passed as dimensional arguments to functions. Here are the operators:

$$(\texttt{first}\ F)_t^{s_0\,s_1\,s_2\cdots s_i\cdots} = F_0^{s_0\,s_1\,s_2\cdots s_i\cdots} \tag{1.32}$$

$$(\texttt{next}\ F)_t^{s_0\,s_1\,s_2\cdots} = F_{t+1}^{s_0\,s_1\,s_2\cdots s_i\cdots} \tag{1.33}$$

$$(F\ \texttt{fby}\ G)_t^{s_0\,s_1\,s_2\cdots} = \begin{cases} F_0^{s_0\,s_1\,s_2\cdots s_i\cdots}, & t=0 \\ G_{t-1}^{s_0\,s_1\,s_2\cdots s_i\cdots}, & t>0 \end{cases} \tag{1.34}$$

$$(\texttt{initial}_i\ F)_t^{s_0\,s_1\,s_2\cdots s_i\cdots} = F_t^{s_0\,s_1\,s_2\cdots 0\cdots} \tag{1.35}$$

$$(\texttt{succ}_i\ F)_t^{s_0\,s_1\,s_2\cdots s_i\cdots} = F_t^{s_0\,s_1\,s_2\cdots (s_i+1)\cdots} \tag{1.36}$$

$$(F\ \texttt{sby}_i\ G)_t^{s_0\,s_1\,s_2\cdots s_i\cdots} = \begin{cases} F_t^{s_0\,s_1\,s_2\cdots 0\cdots}, & s_i=0 \\ G_t^{s_0\,s_1\,s_2\cdots (s_i-1)\cdots}, & s_i>0 \end{cases} \tag{1.37}$$

The main drawback of Field Lucid is that it adopts an "absolute" view of the multidimensionality; the names of the multiple (orthogonal) dimensions are preordained. Thus, it is not possible to apply a function that expects its arguments to be defined over space dimension 0 to arguments defined over space dimension 1.

1.8 Dimensional abstraction: Indexical Lucid

In 1991, Faustini and Jagannathan introduced the language Indexical Lucid [36, 37], which is Lucid with dimensional abstraction. In so doing, all of the multidimensional ideas being worked on were radically simplified.

In Indexical Lucid, new indices can be explicitly created using the `index` declaration within an *indexical where clause*:

```
where
  index a, b;
  ...
end
```

The new indices are a and b and variables may be defined to vary in those indices in addition to the time dimension, the latter always implicit. As predicted in [95], the `is current` declaration is no longer required. The implicit, temporary dimension created by this operation can now be explicitly declared by the programmer.

The standard Lucid operators are available, together with a dimension name suffixing scheme. For example, to access some dimension a, one can use any of `first`.a, `next`.a, `fby`.a, `wvr`.a, `upon`.a, `asa`.a and `@`.a. The time dimension can be accessed via the standard Lucid operators, or by suffixing `.time` to them. It should be noted that Indexical Lucid dimensions vary over the integers, just as does the time dimension. Also, the terms dimension and index are used interchangeably here, but at the time only the word index applied. Later, with the explicit references to multidimensional programming, the indices were renamed dimensions.

This new feature allows functions, and even variables, to be defined with formal dimension parameters. One or more dimensions can be added to a definition, using a suffixing scheme similar to that used for the operators. The dimension names used are not actual dimensions, but placeholders for dimensions supplied when the function is called. Today, we might call them dimensional templates.

An additional ternary indexical operator, `to`, copies one dimension to another for a specific expression: a `to` b x. The expression x varies now in both dimension b and dimension a. The same result could be achieved by writing x `@`.b `#`.a.

As mentioned before, the `#` and `@` operators become `#`.d and `@`.d, where `#`.d allows one to query about *part* of the set of dimensions rather than the *entire* set. Similarly, `@`.d allows one to change some of the dimensions at a time without having to access any of the others, or the whole set at the same time.

The basic operators thus become:

```
first.d X = X @.d 0;
next.d X = X @.d (#.d + 1);
prev.d X = X @.d (#.d − 1);
X fby.d Y = if #.d ⩽ 0 then X else Y @.d (#.d − 1);
X wvr.d Y = X @.d T
            where
              T = U fby.d U @.d (T + 1);
              U = if Y then #.d else next.d U;
            end
X asa.d Y = first.d (X wvr.d Y);
X upon.d Y = X @.d W
             where
               W = 0 fby.d if Y then (W + 1) else W;
             end
```

The prime number tester becomes in Indexical Lucid:

```
$\neg idivn$ asa.$a$ idivn $\vee\ i * i \geqslant n$
where
   index $a, b$;
   $i = 2$ fby.$a\ i + 1$;
   $j = i * i$ fby.$b\ j \geqslant n$;
   $idivn = j \equiv n$ asa.$b\ j + i$;
end
```

The eductive model of computation is still used successfully to run Indexical Lucid. The demand-driven interpreter coupled with the warehouse needs little modification in order to handle the multiple dimensions. From the point of view of the eductor, the dimensional tags just become more complex.

To summarize, dimensions can now be manipulated, exchanged, transposed, etc., but the *rank* of an object — the set of dimensions in which it varies — cannot be accessed. For this, it is necessary to have dimensions as first-class values (see §1.13).

1.9 Going parallel: Granular Lucid

The first attempt at a production version of the full Lucid language took place with the creation of Granular Lucid (GLU) in the early 1990's by Jagannathan and Faustini. GLU was a hybrid parallel-programming system that used Indexical Lucid as coordination language to ensure the parallel execution of coarse-grain tasks written in `C` [47].

In GLU programs, an intensional core (Indexical Lucid) specifies the parallel structure of the application and imperative functions—written in `C`—perform the calculations. This high-level approach was designed to take advantage of coarse-grain data parallelism present in many applications, for example, matrix multiplication, ray-tracing, video encoding, CT-scan reconstruction, etc.

The eduction algorithm was adapted to distribute these tasks across a shared-memory multiprocessor or a network of distributed workstations. Because of the coarse-grain nature of the problems addressed, Indexical Lucid became an efficient programming environment, since the overhead of the eductive algorithm became inconsequential. Furthermore, the runtime system could be compiled for a variety of system configurations.

To deal with `C` operations that might involve side-effects, three binary operators were added to Indexical Lucid [46]: "`,`", "`!`" and "`?`". Expression $(x\texttt{,}y)$ returns the value of y but only after x has been evaluated. Expression $(x\texttt{!}y)$ returns the value of y, after having launched the calculation of x. The difference between these two operators is that "`,`" is total while "`!`" is partial, i.e., it will produce a result even if x is undefined. Expression $(x?y)$ evaluates both x and y and returns the first value computed, corresponding to a parallel merge.

The development of GLU showed that intensional programming can be naturally applied to real-world problems. Intensional programming, coupled with eduction, provided an elegant, straightforward solution for transforming existing imperative programs with latent parallelism into parallel programs, simply by breaking up the imperative tasks into manageable chunks whose recombination could be described in Lucid and whose distribution could be managed by the runtime system. The GLU language and runtime system were used for the second Lucid book, entitled *Multidimensional Programming* [5], published in 1995 by Ashcroft, Faustini, Jagannathan and Wadge.

The GLU system, although interesting from an academic point of view, was not widely used, for one very important reason; the reworking of existing imperative applications required digging through existing `C` code to determine how best to distribute tasks, and this was a non-trivial activity.

1.10 Possible-worlds versioning: The context comes to the fore

At the same time as the work on Indexical Lucid and GLU was taking place, author Plaice and Bill Wadge began to work on the problem of *possible-worlds versioning*, in which the very *structure* of a computer program varies with a multidimensional context. Although it was not understood at the time, this work would have tremendous influence on the future developments of Lucid.

The concept of possible-worlds versioning was first presented in [77]. A universe of possible worlds was defined *in an algebraic manner*, with a partial order defined over that universe. The structure of a program, or of any other hierarchically defined entity, was an intension. Building a specific extension in a specific possible world simply meant using the most relevant versions of each component, including of build files used to define how components are to be assembled together. The version tag for the resulting built component was the least upper bound of the version tags of all the chosen source components.

In each possible world, a computer program's structure can be different, and each component of the program can be different. When a component is requested, then the *most relevant version* of that component is chosen, and building of the system continues with lower level components.

A detailed discussion of possible-worlds versioning, complete with a presentation of a number of experiments in versioned electronic documents, file systems, Web pages and other electronic media, is given in "Possible Worlds Versioning", by the first two authors, also found in this volume [59].

For the purposes of the presentation below, it became clear, as discussions took place between the various researchers involved, that the context is an active entity that permeates both a program's structure and its behavior. It would also become clear that the values held by all of the dimensions in a Lucid program formed such a context, and that it should become a first-class entity (see §1.14).

1.11 Versioned definitions: Plane Lucid

While the general problem of possible-worlds versioning was being studied, so was the more specific problem of adding versioned definitions to Lucid. In particular, Du and Wadge developed Plane Lucid [32, 31], in which multiple definitions can be given for the same variable. When a (*identifier*, *context*) is requested, then the *most relevant* definition for the identifier, with respect to the current context, must be chosen before that definition may be evaluated.

Du and Wadge used Plane Lucid to define a three-dimensional *intensional spreadsheet*. Just as in a regular spreadsheet, the intensional spreadsheet uses two spatial dimensions, but adds a third, temporal dimension. Each cell is indexed by a triple (h, v, t). The spreadsheet is viewed as a single entity whose value varies according to the context. The value in a specific context may be defined in terms of values in other contexts.

Plane Lucid is a language similar to Field Lucid, with additional intensional operators for navigating the space and time dimensions. Each of the three dimensions receives five context-switching operators. These have simple Indexical Lucid equivalents. For example, Table 1.1 lists the operators for the horizontal dimension together with their Indexical Lucid counterparts.

Table 1.1: Indexical Lucid equivalents to horizontal Plane Lucid operators

Plane Lucid operators	**Indexical Lucid equivalents**
`side` A	A `@.`h 0
`right` A	A `@.`h (`#.`$h + 1$)
`left` A	A `@.`h (`#.`$h - 1$)
A `hsby` B	`if #.`$h > 0$ `then` B `@.`h (`#.`$h - 1$) `else` A `fi`
A `hbf` B	`if #.`$h < 0$ `then` A `@.`h (`#.`$h + 1$) `else` B `fi`

Since not every cell has the same definition, Du and Wadge specified four definition levels for values of cells. They are, in decreasing order of priority: local, dimensional, planar and global. As might be expected, they represent progressive generalizations. For example, in spreadsheet S:

$$\begin{aligned}
&S[h \leftarrow 3, v \leftarrow 4, t \leftarrow 5] = 3;\\
&S[v \leftarrow 4, t \leftarrow 5] = 2;\\
&S[t \leftarrow 5] = 1;\\
&S = 0;
\end{aligned}$$

cell $(3, 4, 5)$ is locally defined to take the value 3; cells $(?, 4, 5)$ are dimensionally defined to take the value 2; cells $(?, ?, 5)$ are planarly defined to take the value 1; and all other cells are given a default value of 0.

1.12 List dimensions: Attributes and functions

In Indexical Lucid, a dimension can only take integers for values. This is natural, as the objects being manipulated by Indexical Lucid are assumed to be multidimensional arrays, and the integer tuples can be used to index into these arrays. However, there are other kinds of data structure that require different kinds of index. In particular, a finite tree can naturally be understood as a mapping from tuples — finite lists — to values.

This intuition was taken up by Senhua Tao [90], under Wadge's supervision, to treat attribute grammars — used for defining the syntax and semantics of programming languages — intensionally, in an effort to properly support circular attribute definitions.

Tao created TLucid, an extension of Lucid with a list-valued dimension, encoding the position inside a syntax tree, in addition to the standard time dimension. The length of the finite list corresponds to the length of the path to the corresponding node in the tree. Context-switching operators are added to TLucid for traversing this new dimension. These operators are given in Table 1.2, along with equivalents in an "Indexical Lucid" that would allow non-integer dimensions.

Table 1.2: "Indexical Lucid" equivalents to TLucid operators

TLucid operators	**"Indexical Lucid" equivalents**
`index`	`#`$.i$
`root` N	N `@`$.i$ nil
`parent` N	N `@`$.i$ $(tl$ `#`$.i)$
`nextsib` N	N `@`$.i$ $(((hd$ `#`$.i) + 1) : (tl$ `#`$.i))$
`child`(N, k)	N `@`$.i$ $(k :$ `#`$.i)$

The same technique, this time in a multidimensional framework, was used by Panagiotis Rondogiannis and Wadge [81, 82, 16] to implement higher-order functions in Lucid itself. In these articles, they simulate the calling structure of a higher-order program by using list-valued dimensions indicating which of each of the functions has been called and from where. The advantage of this approach is that no closure operations are required. However, the technique is not generally applicable to partially applied functions.

1.13 First-class dimensions: Multidimensional Lucid

The natural next step was to introduce dimensions as first-class values, work undertaken by Joey Paquet and author Plaice in Tensor Lucid [66, 68], developed to write tensor equations naturally. By allowing declared dimensions to be used as ordinary values, the total dimensionality of an object becomes directly accessible. However, all dimensions must still be created lexically, and cannot be created on demand, during execution, and defining values as dimensions was the next step.

Paquet's work was simplified by author Plaice in Multidimensional Lucid [71], in which any ground value may be used as a dimension. Multidimensional Lucid is simply ISWIM with two new primitives, indexical query ($\texttt{\#}E$) and context change ($E\ \texttt{@}E_1\ E_2$). In $\texttt{\#}E$, expression E is evaluated to a value v and the context is then queried, using v as the dimension. In $E\ \texttt{@}E_1\ E_2$, expression E_1 is evaluated to v_1 and E_2 is evaluated to v_2; then E is evaluated in the current context, modified so that dimension v_1 yields the value v_2. The difference with Indexical Lucid is that the dimensions need not always be identifiers; therefore they can be created on the fly, opening up many new possibilities.

The example program is here shown in Multidimensional Lucid:

$$\begin{array}{l}
n = \mathit{input};\\
i = 2\ \texttt{fby}.0\ i+1;\\
j = i*i\ \texttt{fby}.1\ j+1;\\
\mathit{idivn} = j \equiv n\ \texttt{asa}.1\ j \geqslant n;\\
\mathit{output} = \neg \mathit{idvin}\ \texttt{asa}.0\ \mathit{idivn} \vee i*i \geqslant n;
\end{array}$$

The semantics of Multidimensional Lucid was much simpler than that of Indexical Lucid. However, since dimensions could be created on the fly, the eductive algorithm used since the first implementation of Original Lucid was no longer applicable, because the potential wastage of memory while caching partial results was essentially unbounded. Until this problem could be resolved, developing an interpreter for Multidimensional Lucid would have been of limited utility.

1.14 First-class contexts

The idea of contexts as first-class values in Lucid came from two separate directions. As mentioned in §1.10, one was the work in possible-worlds versioning, leading to an understanding of a context as an entity permeating the behavior and structure of a program or of a set of programs. The second came from Joey Paquet, in the development of the GIPSY project that he is leading in developing a generic infrastructure for the interpretation and compilation of Lucid variants [67].

In 2001, Paquet sent an email to his Concordia University colleague Peter Grogono and to author Plaice, with a proposal for generalizing the work on dimensions as values. He suggested that first-class contexts be added to Lucid. The proposal was that the @ operator should become binary: $E'\ \texttt{@}\ E$ should mean that E should evaluate to a context, and that E' would then be evaluated, using E as its new context. In that letter, he also proposed that perhaps the E could even evaluate to a *set of contexts*, which would mean that E' would be evaluated in each context within that set, yielding a set of values.

Notwithstanding the possible advantages of adding contexts and sets of contexts to Lucid, the implementation issues of such an addition are daunting, because the problems outlined in the previous section would simply be amplified.

1.15 Cartesian programming: TransLucid

The TransLucid project started in late 2005, with a visit to UNSW by Bill Wadge. During his stay, he explained an earlier idea that he and Tony Faustini had had about *lazy eduction*, in which requests to the warehouse caching the partial results would be incremental. When requesting an (*identifier*, *tag*) pair, the execution engine would begin with an empty tag, and the warehouse would either return a result or a request for information about additional dimensions, in which case a new request would need to be made with a more refined tag. Once sufficient information was provided, then the warehouse could provide the answer or provoke a calculation, if necessary.

With this idea, an initial set of rules for a language with contexts as values was developed, as outlined in the previous section. This language, presented in author Ditu's PhD thesis [28], had as goal to be usable as a real programming language, either directly or as the target language for compiling languages in a variety of paradigms. An example problem is the Ackermann function:

$$\begin{aligned} ack = {} & \texttt{if } \#0 \equiv 0 \texttt{ then } \#1 + 1 \\ & \texttt{elsif } \#1 \equiv 1 \texttt{ then } ack \texttt{ @ } [0 \leftarrow \#0 - 1, 1 \leftarrow 1] \\ & \texttt{else } ack \texttt{ @ } \big[0 \leftarrow \#0 - 1, 1 \leftarrow ack \texttt{ @ } [1 \leftarrow \#1 - 1]\big] ; ; \end{aligned}$$

The use of $[\cdots]$ allows the specification of a new context relative to the current context, replacing the values for an arbitrary set of dimensions. As in GLU, new types and operators can be added to the language.

With TransLucid, there are two kinds of data structure: the explicit tuple, and the implicit infinite multidimensional array, which we call a *hyperdaton*. When used as a context, the explicit tuple corresponds exactly to a multidimensional coordinate into the hyperdaton. For this reason, we have coined a new term for programming in TransLucid: *Cartesian programming*, with the clear allusion to the Cartesian referential system. In some sense, everything has become simpler. We have reached, in the words of Jean Dhombres [25], « La banalidad del referencial cartesiano » ("The trivialitity of the Cartesian referencial system").

Since one can always use more parameters—dimensions—as needed to describe a problem, it is possible to translate other programming paradigms into TransLucid, in order to have a single intermediate language. It is also possible to go the other way, and to add some of the multidimensional features of TransLucid to other languages.

The development of TransLucid is continuing, with many experiments in implementation, which are highly relevant to the original discussion in the introduction, which related the importance of architecture design to the development of programming languages. In his honours thesis [79], Toby Rahilly presented a multithreaded eduction engine, and in so doing, derived the idea of *lazy tuples*, in which the values associated with the dimensions in a context are only calculated if necessary. The implementation led, not simply to faster running programs, but also to a more powerful programming model. With lazy tuples, it will be possible to extend TransLucid to manipulate infinite contexts.

At the semantic level, work needs to be done so that TransLucid can be integrated with other aspects of a computer system, in order for it to be usable with files, networks, databases, stream I/O, memory-mapped I/O, and so on. To do this will require adding sets of contexts, types and expressions as values, along with means to refer to time and concurrency. In time, a TransLucid system, with an evolving set of declarations, definitions and demands, will correspond to an object in an object-oriented environment.

1.16 Conclusions

The initial Lucid publications go back to some 30 years ago, with the attempts by Bill Wadge to formalize existing computation for the purposes of formal program verification. Since then, the Lucid programming language has undergone many changes, leading to the TransLucid language of today. These changes were not mere technical sleights of hand: most of them required a radical reinterpretation of the very concept of computation, leading each time to a deeper, yet simpler, understanding. At each of these stages, Bill Wadge has been present, with ideas for his students and his colleagues. We are all richer for it, as is the discipline.

The current development of a wide variety of manycore and multicore architectures makes the current research in TransLucid of wide relevance. Successful deployment of declarative programming is of greater importance today, where high-performance computing becomes mainstream, than it ever has been.

Chapter 2

Possible-Worlds Versioning

(with Blanca Mancilla)
Mathematics in Computer Science 2(1):63–83, 2008
`http://dx.doi.org/10.1007/s11786-008-0044-8`

We present a history of the application of the possible-world semantics of intensional logic to the development of versioned structures, ranging from simple software configuration to the networking of distributed context-aware applications permeated by multiple shared contexts.

In this approach, all of the components of a system vary over a uniform multidimensional version, or context, space, and the version tag of a built version is the least upper bound of the version tags of the selected bestfit components. Context deltas allow the description of changes to contexts, and the subsequent changes to components and systems from one context to another. With æthers, active contexts with multiple participants, several networked programs may communicate by broadcasting deltas through a shared context to which they are continually adapting.

2.1 Introduction

This paper presents the development of *possible-worlds versioning*, from 1993 to the present, and its application in the areas of software configuration, hypertext, programming language design and distributed computing. This research has followed naturally from the seminal article by author Plaice and William (Bill) Wadge, "A New Approach to Version Control" [77], which demonstrated that software configuration could be understood, and greatly simplified, using the possible-worlds semantics of Richard Montague's intensional logic, and which presented the vision of a flexible Unix-like system called *Montagunix*.

The application of possible-worlds versioning has been used successfully — as will be seen in this article — to add versions to C programs, electronic documents, file systems, Linux processes and programming languages, among others, and to create collaborative software for shared browsing of Web sites. In the 2004 *ACM Hypertext* conference, Doug Engelbart told the authors that this collaborative work [73] was "exactly what we were trying to do in the 1960s" [33].

The intuition driving the above research has changed over time, even though the technical solutions have been remarkably stable. Initially, when possible-worlds versioning was simply being applied to the problem of software configuration, the science fiction-like intuition of possible worlds sufficed: in each world, there is a completely new copy of every entity. This copy may be similar to corresponding copies in other worlds, but it is a complete copy. The technical details and implementations then followed from this vision.

Subsequently, as the problems being addressed became more dynamic in nature—as, for example, in shared browsing of a Web site—the possible world was more easily understood as a physical medium permeating a system and every component therein, as water does to cells in a living body. Any change to the medium can affect instantaneously any part of the system; conversely, any part of the system may change the medium, indirectly affecting other components of the system. In Marshall McLuhan's words, "Environments are not simply containers, but are processes which change their content entirely" [61, p.275].

The problems needing to be resolved today will require a more refined understanding of this permeating medium. If one looks at the Web today, one can see the rise of many different kinds of *communities*, for sharing ideas, software, entertainment, etc. This Web is neither a sea of atomized individuals nor one giant shared space: there are *many* shared spaces, and each individual might be participating in a number of different, overlapping spaces.

This complex arrangements of spaces is also reminiscent of science fiction, but on a higher plane, where one can pass from one virtual space to another, while still retaining some part of the original space: the potentialities are mindboggling, especially when there are many overlapping *universes* of possible worlds, so that one may be simultaneously in different worlds in different universes. This vision, of course, corresponds exactly to present-day needs for the interactions between mobile and ubiquitous computing.

Bill Wadge has been a key player in developing these intuitions. Of course, many of the projects described below have been undertaken by students under his supervision. More importantly, however, he has always been present for the discussions about the future of the research. It was he who discovered the relevance of the ideas of visionaries such as Marshall McLuhan and Ted Nelson — as well as the ancient philosophical debate between atomism and plenism — to our work.

The article is arranged as follows. We begin (§2.2) with an exposition of intensional logic, possible-worlds semantics and the interplay between intension and extension. We then show (§2.3) that possible-world semantics is applicable to the structure of programs. We continue with experiments in electronic documents (§2.4) and in versioned programming (§2.5). The idea of *intensional communities* leads naturally to the æther and its applications (§2.6). The concluding remarks propose extending these solutions, using synchrony, to wide-scale distributed computing.

2.2 Background: Possible worlds and intensional logic

In 1987, Bill Wadge and Tony Faustini published the article "Intensional Programming" [35], which showed that the possible-worlds semantics of intensional logic was directly applicable to the semantics of Lucid, especially its multidimensional variants. The semantics of a variable is an intension, a mapping from possible worlds to specific values. In each possible world, when one refers to the values of other variables, one is referring to the values within the current world, *unless* some other world is explicitly mentioned.

In this section, we present a brief history — inspired directly from Bill Wadge's article, "Intensional Logic in Context" [93] — of the ideas of possible-world semantics and of intensional logic.

Ever since the beginnings of logic, it has been understood that there is a difference between sentences that are necessarily true because of the nature of logic and sentences that just happen to be true because of contigent factors. For example, the sentence:

Nine is a perfect square. (2.1)

and the sentence:

The number of planets is nine. (2.2)

were once both true, but the nature of the truth in the two sentences is different, and they cannot be substituted equivalently into other sentences, as in:

Kepler believed that nine is a perfect square. (2.3)

and:

Kepler believed that the number of planets is nine. (2.4)

Given that Johannes Kepler (1571–1630) was a gifted mathematician and remarkable astronomer, and that only six planets were known during his lifetime, likely (2.3) is true and (2.4) false.

Furthermore, in August 2006 the International Astronomical Union (IAU) re-classified Pluto as a "dwarf planet," reducing the number of classic planets to eight. In the light of this statement, the sentence deduced from sentences (2.1) and (2.2):

$$\textit{The number of planets is a perfect square.} \qquad (2.5)$$

is no longer true either.

The existence of this general problem was recognized by Aristotle (384–322 B.C.E.), *The Logician*, "the first to state formal laws and rules" for logic [10, p.19]. In his writings on formal logic, collectively known as the *Organon*, Aristotle introduced modal logic and distinguished between two modes of truth: *necessity* and *contigency* [10, pp.55-6].

Because of the comprehensive nature of Aristotle's work, his writings dominated all study of logic in the Western world until the development of mathematical logic, beginning in the late seventeenth century. Nevertheless, developments did take place in the understanding of modalities, often with respect to theological arguments as to the nature or existence of God and the world.

In particular, John Duns Scotus (1265/6–1308), a Franciscan scholar, introduced the concept of *possible worlds*. In opposition to Thomas Aquinas, Scotus asserted the primacy of *Divine Will* over *Divine Intellect*: the existing world does not exist because of some moral necessity, but, rather, because of a divine choice. The world we live in could be different and it is just one of numerous logically consistent possible worlds. Gottfried Wilhelm von Leibniz (1646–1716), the founder of mathematical logic [11, p.258], put forward that "a necessary truth must hold in all possible worlds" [14, p.10].[1]

As mathematical logic post-Leibniz developed, with a strong anti-Aristotelian bias, the modalities were pushed aside, at least temporarily, and sentences were designated as being either true or false. The focus of attention was placed entirely on mathematical rigor. In so doing, the distinction between *intension* and *extension* surfaced.

Bertrand Russell wrote (1903) [11, p.361]: "*Class* may be defined either extensionally or intensionally. That is to say, we may define the kind of object which is a class, or the kind of concept which denotes a class: this is the precise meaning of the opposition of extension and intension in this connection." But, he believed "this distinction to be purely psychological." For Russell, the difference between extension and intension is quantitative, not qualitative: the intension is only needed because one cannot write down infinite classes.

However, the distinction between the two is qualitative as well. Already in the 3rd century, Porphyry of Tyre noted the difference between *genus* and *species* in his *Isagoge* [11, pp.135,258].[2] More recently, the *Logique de Port-Royal* (Antoine Arnault et Pierre Nicole, 1662), distinguished *compréhension* and *étendue*, and Leibniz retained these terms. Leibniz called the "*comprehension* of an idea the attributes which it contains and which cannot be taken from it without destroying it." He called "the *extension* of an idea the subjects to which it applies" [11, p.259].

Rudolf Carnap (1891–1970), in *Meaning and Necessity*, made explicit the connection between Leibniz's possible worlds and the intension-extension duality. He began with a first-order system S_I with standard connectives and quantifiers. A *state description* is a class of sentences in S_I "which contains for every atomic sentence either this sentence or its negation, but not both, and no other sentences." The description "obviously gives a complete description of a possible state of the universe of individuals with respect to all properties and relations expressed by predicates of the system. Thus the state-descriptions represent Leibniz' possible worlds" [14, p.9].

Carnap then distinguished between *truth* and *L-truth*. *Truth* simply means truth with respect to a specific state description, and *L*-truth means truth in all state descriptions, using the conventions that two *predicators* (predicate symbols) have the same extension if, and only if, they are equivalent and the same intension if, and only if, they are *L*-equivalent.

[1]Nevertheless, he too used logic to participate in theological arguments. In his *Essais de Théodicée sur la bonté de Dieu, la liberté de l'homme et l'origine du mal* (1710) he stated that notwithstanding the many evils of this world, "We live in the best of all possible worlds." François Marie Arouet de Voltaire (1694–1778) replied in his *Candide ou l'optimisme* that "If this is the best of all possible worlds, what then are the others?"

[2]These terms are kept in today's classification of life: *genus* being a taxonomic grouping of organisms and containing several *species*, the latter designating a group of organisms capable of interbreeding.

To give an overall semantics to this process, Carnap stated that “an assignment is a function which assigns to a variable and a state-description as arguments an individual constant as value” [14, p.170]. As a result, an intension becomes a mapping from the state-descriptions to its extensions. According to Dowty [30, p.145], Carnap’s “intension is nothing more than all the varying extensions (denotations) the expression can have, put together and *organized*, as it were, as a function with all possible states of affairs as arguments and the appropriate extensions arranged as values.”

Saul Kripke (1940–) went further and developed possible world semantics for modal logic, in which the possible worlds are indices. “The main and the original motivation for the *possible world analysis* — and the way it clarified modal logic — was that it enabled modal logic to be treated by the same set theoretic techniques of modal theory that proved so successful when applied to extensional logic. It is also useful in making certain concepts clear” [50, p.19].

For Kripke, a possible world is “a little more than the miniworld of school probability blown large” [50, p.18]. Kripke does not attempt to define a “complete counterfactual course of events” arguing further that there is no need to do so: “A practical description of the extent to which the ‘counterfactual situation’ differs in the relevant way from the actual facts is sufficient.” This will prove particularly useful later on when describing and formalizing only the changes to the context (and not the entire context) in a running system, changes that need “broadcasting.”

According to Dowty [30, p.145] “with the advent of Kripke’s semantics for modal logic (taking *possible worlds* as indices), it became possible for the first time to give an unproblematic formal definition of *intension* for formalized languages.” Soon after, Richard Montague (1930–1971) created his *intensional logic*, culminating in his paper “The Proper Treatment of Quantification in Ordinary English” [91].

All this was generalized by Dana Scott in his 1969 paper “Advice on Modal Logic” [83]. He assumed a nonempty set I of reference points that do not require any accessibility relation, just like in possible worlds semantics. The truth value of a sentence ϕ at a particular point is the *extension* of ϕ at that point. The intension of ϕ is an element of 2^I, a function that maps each point i to the extension of ϕ at i. A (unary) *intensional operator* is a function mapping intensions to intensions, i.e., an element of $2^I \to 2^I$. Bill Wadge quoted the following prescient passage:

> This situation is clearly situated where I is the context of time-dependent statements; that is the case where I represents the instants of time. For more general situations one must not think of the $i \in I$ as anything as simple as instants of time or even possible worlds. In general we will have
>
> $$i = (w, t, p, a) \tag{2.6}$$
>
> where the index i has coordinates; for example w is a world, t is a time, $p = (x, y, z)$ is a (3-dimensional) position in the world, a is an agent [*this is 1969!*], etc. All these coordinates can be varied, possibly independently, and thus affect the truth of statements which have indirect references to these coordinates.

The simplicity of Scott’s approach is directly applicable to computing. It has been used, and continues to be used, with success for the development of the Lucid language [74]. In this article, we take the idea and apply it to systems whose structure and behavior change as the context in which they are placed changes.

2.3 Possible-worlds versioning

In 1993, Bill Wadge and author Plaice published an article entitled “A New Approach to Version Control” [77], which applied the use of possible-world semantics[3] to the creation of software families. We summarize the results here.

[3]Originally, the term used for this work was called *intensional versioning.* However, in the software configuration community [18], intensional versioning has become a consecrated term for any form of automatic variant selection, but without any reference to possible-worlds semantics. For this reason, we now refer to *possible-worlds versioning.*

2.3.1 Introduction

In his 1974 seminal paper on software configuration [69], David Parnas described a need for software families. In his vision, a family should contain many different pieces of software, all slightly different. These are now called *variants*.

Wadge and Plaice simply took the intuition of *possible world*, in which there is a *complete state of affairs*, and assumed that in each such world, there was a version of every component and every system. From this intuition, it follows naturally that a software family is an intension and the individual variants are extensions.

However, a basic software engineering principle is to have a single canonical copy of every entity and to avoid duplicate entities (i.e., their variants) in order to prevent unnecessary branching, a very error-prone process. The question remained: How could possible-world semantics be used to provide maximum sharing of code across the variants?

The solution provided was to define a *uniform version space*, shared by all components of a system, and to define thereon a partial order defined over that space. All components are understood to vary conceptually across the entire version space. Physically, on the other hand, any given component may only come in a very limited number of versions, thereby avoiding the unrealistic supposition that the latest version has been developed for each and every component.

When a specific version of the system is to be built, then the *most relevant*, or *bestfit* version, with respect to the partial order, of each component is selected for the build process. This approach is called the *variant substructure principle*. By default, the refined versions inherit from coarser versions; this is called *version inheritance*.

Components here mean not only pieces of codes, but also any other software (or hardware) component, such as splash screens, drop-down menus, build files, configuration files, documentation, i.e., whatever might constitute part of the final deliverable.

2.3.2 Basic definitions (1)

The definitions in this section and in section 2.4.2 do not correspond exactly to the actual definitions used in the articles being summarized. Rather, the key ideas of *context*, *versioned object* and *bestfit version* are being defined so that the discussion below can be understood.

Definition 1. *A* context $\kappa \in \mathbb{K}$ *is a mapping*

$$\kappa \quad ::= \quad (d : v)^+$$

where d is a dimension *and v is a* value.

To access the value associated with a dimension d, we write $\kappa(d) = v$. If $\kappa(d)$ is undefined, we write $\kappa(d) = \bot$. The empty context is written ϵ. The domain of κ, written dom κ, consists of the set of dimensions for which $\kappa(d) \neq \bot$.

Definition 2. *Context κ is* less refined *than context κ', written $\kappa \sqsubseteq \kappa'$, when $\forall d \in \text{dom}\, \kappa, \kappa(d) = \kappa'(d)$.*

Definition 3. *Contexts κ and κ' are* consistent *if $\forall d \in \text{dom}\, \kappa \cap \text{dom}\, \kappa', \kappa(d) = \kappa'(d)$.*

Definition 4. *The* join *of two consistent contexts κ and κ', written $\kappa + \kappa'$, is the union of κ and κ'.*

Definition 5. *Let $\mathbb{A}$ be a set of objects. A* versioned object $\mathcal{A}$ of type $\mathbb{A}$ *is a set of pairs $\mathcal{A} = \{(\kappa_1, \alpha_1), \ldots, (\kappa_n, \alpha_n)\} \subseteq \mathbb{K} \times \mathbb{A}$.*

The domain of $\mathcal{A}$, written dom $\mathcal{A}$, is the set $\{\kappa_1, \ldots, \kappa_n\}$. If $(\kappa, \alpha) \in \mathcal{A}$, we write $\mathcal{A}(\kappa) = \alpha$.

Definition 6. *Let $\mathcal{A}$ be a versioned object and let κ_{req} be a context. Then the* bestfit version of $\mathcal{A}$ with respect to the *requested context κ_{req} is $\big(\kappa_{\text{best}}, \mathcal{A}(\kappa_{\text{best}})\big)$, where:*

$$\kappa_{\text{best}} = \max\{\kappa \in \text{dom}\, \mathcal{A} \mid \kappa \sqsubseteq \kappa_{\text{req}}\}$$

The version tag is kept alongside the component in order to be able to compute the version tag of higher-level components, as seen below.

2.3.3 Experiment: Software configuration

Plaice and Wadge validated this approach by taking a C programming environment, Sloth [78], and adding versions transparently to create Lemur. With Lemur, creating new variants of a piece of software was radically simplified, in comparison to the standard methodology at the time.

The Sloth system is used to build C *modules* and each is held in a directory. A number of component files within the directory are used to automatically generate a C file named `prog.c` for that module. Any component file can come in several versions, each encoded as a *tag* at the filename level in the filename. When a version of the module is requested, the most relevant version of each component file is chosen. In the end, the actual built version of `prog.c` is the least upper bound of the chosen versions of the component files.

A Sloth C module can import other modules using an *import list*. This import list can itself be versioned, using the mechanisms described above. But it is possible to go further: for each named component in the import list, a new requested version may be attached. For each of these components, Sloth proceeds with the new requested version, and returns the best possible version of the built subsystem. The versions of these subsystems then contribute to the version tag of the whole system.

We present a cleaned-up version of Lemur. A system contains a number of directories at the same level, and each directory contains one module, whose name is that of the directory. In each directory, there are three kinds of file:

- import_κ are tagged import list files;
- header_κ.i are tagged header files;
- body_κ.i are tagged body files.

Each import file contains a sequence of names of other directories. When version κ_{req} of module m_0 is requested, files prog_κ_{m_i}.c and dep_κ_{m_i} are created in each of a set of directories $\{m_0, m_1, \ldots, m_n\}$, as follows.

Let m be a requested module and κ_{req}. Then there will be:

- a bestfit import list file import_κ_{i_m}
- a bestfit header file header_κ_{h_m}.i
- a bestfit body file body_κ_{b_m}.i

If file import_κ_{i_m} is empty, then $\kappa_m = \kappa_{i_m} + \kappa_{h_m} + \kappa_{b_m}$, and file dep_$\kappa_m$ contains:

#include header_κ_{h_m}.i

and file prog_κ_m.c contains:

#include dep_κ_m
#include body_κ_{b_m}.i

The file dep_κ_m will be used by modules importing module m.

If file import_κ_{i_m} is non-empty, then there will be a list of modules $p_1, \ldots, p_r$ in that file. For each p_j, files dep_κ_{p_j} and prog_κ_{p_j}.c are created. Then $\kappa_m = \kappa_{i_m} + \kappa_{h_m} + \kappa_{b_m} + \sum_{j=1..r} \kappa_{p_j}$. File dep_$\kappa_m$ merges the r dependency files dep_κ_{p_j} and adds at the end:

#include header_κ_{h_m}.i

File prog_κ_m.c contains:

#include dep_κ_m
#include body_κ_{b_m}.i

Once all of the files prog_κ_m.c are created, then they must be compiled and linked together. The Lemur system worked remarkably well, with the caveat that a system could only include a single version of each module that was included. This was not so much a restriction of the versioning process but, rather, of the fact that the C code would have had to be created dynamically to ensure that there would be no name clashes for different versions of the same module.

2.3.4 Discussion

Possible worlds versioning tremendously simplifies configuration of any system (or of parts of it) in which the components have high variability. Configuration of the system at any requested instant (i.e., at any specific version) is automatic, since version κ of a compound entity is the result of combining version κ of each of its parts. A system can be an application with many components, whose version at any time can be assembled without human interaction. And since the configuration is demand-driven, along with version inheritance, it is possible to have very large version spaces without tying up resources by creating and storing unwanted versions. Version inheritance makes the uniform version space practical, since it allows different versions of the system components to share code and data transparently.

This section emphasized software families, where the components and subcomponents are mainly software objects, whose version space is limited. As should already be clear from the discussion, the components are not limited to software objects and the final system can be anything, even Web pages. Assuming appropriate syntax and run-time support, any entity can be versioned and *adapt itself* to any context, changing its internal structure as the context it resides in changes.

2.4 Navigation through possible worlds

In the mid-1990s, Bill Wadge and his students Taner Yıldırım, Gordon Brown and (Monica) m.c.schraefel applied the ideas of possible worlds versioning to the navigation of the Web and to the creation of families of interlinked Web pages [96, 98, 12, 54]. We summarize the results here.

2.4.1 Introduction

The original article on possible worlds versioning, as discussed in the previous section, was published in March 1993, when the World Wide Web was becoming widely known. At that time, most Web sites consisted of purely static pages, all produced manually. However, there were a few innovative sites that did things differently: By using clickable images and other dynamic forms of linking, pages could be generated on the fly with new or modified information. These generated pages were really "families of pages," equivalent to the families of software just described.

These pages were reproducing a phenomenon little known outside specialized circles at the time: an electronic document is *recreated* every time it is viewed. Just as the philologist claims that a text is different every time it is read because each reader's approach creates a new reading context, the electronic document is different every time it is viewed because it will be rendered for a specific viewing context. It is therefore natural to consider an electronic document to be an intension, and that the extensions are the particular renderings.

What is new with the Web and other hypertext environments is that one is *continually* changing the context. Every click on a button, a clickable region or a hyperlink creates a new context, and the page must be changed accordingly. Furthermore, hyperlinks to neighbouring pages must be regenerated, because the neighbours are all slightly different, to take into account the new context.

This new situation is more dynamic than that in the previous section. One is no longer building isolated entities for a specific possible world but, rather, building entities for a new requested world, given an existing current world, where the new world can be accessed by changing the values of some of the parameters, or dimensions, defining the current world.

2.4.2 Basic definitions (2)

The key new concept defined here is the *context delta*, used to manipulate contexts, and the *delta-versioned object*.

Definition 7. *A* context delta $\delta \in \mathbb{D}$, *or* delta *for short, is an operator for changing contexts:*

$$\begin{aligned} \delta &::= \kappa \mid \overline{\kappa} && (2.7)\\ \overline{\kappa} &::= (d : \overline{v})^+ && (2.8)\\ \overline{v} &::= \mathtt{clear} \mid \mathtt{set}(v) && (2.9) \end{aligned}$$

When $\delta = \kappa$, it is absolute*, and when $\delta = \overline{\kappa}$, it is* relative*. The application of δ to κ_0, written $\kappa_0\delta$, is given by:*

$$\begin{array}{lrcl} \delta = \kappa : & (\kappa_0\delta)(d) & = & \kappa(d) \\ \delta = \overline{\kappa} : & (\kappa_0\delta)(d) & = & \begin{cases} v, & \overline{\kappa}(d) = \mathtt{set}(v) \\ \perp, & \overline{\kappa}(d) = \mathtt{clear} \\ \kappa_0(d), & \textit{otherwise} \end{cases} \end{array}$$

Definition 8. *Let $\mathbb{A}$ be a set of objects. A* delta-versioned object $\overline{\mathcal{A}}$ of type $\mathbb{A}$ *is a set of pairs $\overline{\mathcal{A}} = \{(\delta_1, \alpha_1), \ldots, (\delta_n, \alpha_n)\} \subseteq \mathbb{D} \times \mathbb{A}$.*

Delta-versioned objects are a generalization of versioned objects. The domain of $\overline{\mathcal{A}}$, written dom $\overline{\mathcal{A}}$, is the set $\{\delta_1, \ldots, \delta_n\}$. If $(\delta, \alpha) \in \overline{\mathcal{A}}$, we write $\overline{\mathcal{A}}(\delta) = \alpha$. Best-fitting with delta-versioned objects takes place with respect to a current context, by creating a versioned object first.

Definition 9. *Let $\overline{\mathcal{A}} = \{(\delta_1, \alpha_1), \ldots, (\delta_n, \alpha_n)\}$ be a delta-versioned object and let κ_{cur} be a context. The* versioned object $\mathcal{A}$ generated from $\overline{\mathcal{A}}$ in the current context κ_{cur} *is given by: $\mathcal{A} = \{(\kappa_{\text{cur}}\delta_1, \alpha_1), \ldots, (\kappa_{\text{cur}}\delta_n, \alpha_n)\}$.*

2.4.3 Experiment: Intensional HTML

Taner Yıldırım designed the original IHTML as part of his MSc work [98] by adding *versioned links* to HTML. These links are analogous to the versioned imported components in Sloth modules: they specify a version to use on the file specified by the URL. The versioned links of an IHTML1 page are of the form:

$$\texttt{<a href=}\mathit{URL}\,?\; s_1 = v_1 \,\texttt{\&}\ldots\texttt{\&}\, s_n = v_n\texttt{>} \qquad (2.10)$$

where s_i are string-valued dimensions and the associated values v_i are scalars, either strings or integers.

The IHTML1 implementation was done entirely using CGI, passing the context encoded in the URL to a Perl script. The context becomes an associative array mapping dimensions to values. When the Web page is requested for a particular context, the server converts the IHTML page into an ordinary HTML page, rendered on the fly and adapted to the current context (specified in the URL). If one of the versioned links in the rendered page is followed, then the new context will be the current context *modified* by replacing the values of dimensions that are designated in the versioned link. Unlike in Sloth import lists, links are *relative* to the current context.

As an example, Yıldırım produced a family of Web pages in which one can vary background and foreground color as well as presentation language (English or Turkish). The implementation was quite simple and the family of Web pages was reachable through a fixed main page. Nevertheless, the ideas proved fruitful.

Gordon Brown took this work and for his own MSc redesigned and reimplemented IHTML as a plug-in for the Apache server under Linux, and created IHTML2 [12]. This version allowed for far more complex manipulation of the context in a page because he changed the server-side processing, implementing a module called httpd for the Apache Web server.

Brown's robust implementation made possible some ambitious sites, the most complex (at 4 million pages) being the French sentence builder designed by Bill Wadge and his wife, Christine Wadge, who teaches in the French Department of the University of Victoria, Canada.

At the markup level, IHTML2 includes versioned forms of the HTML tags referring to other documents or files: `a`, `img`, `form` and `frame`, each allowing attributes `version` (absolute delta) or `vmod` (relative delta). These links are called *versioned hyperlinks*.

For anchor tags, the `href` attribute becomes optional, and can be used to link to a different version of the same page. For example:

```
<a vmod="language=French">
```

is a link to the current page, where the current context has been modified so that the `language` dimension is set to `French`. On the other hand

```
<a version="language=French">
```

links to the same page, with a completely new context. In both cases, when this anchor is clicked upon, the requested delta δ_{req} is applied to the current context κ_{cur} to create a new requested context $\kappa_{\mathrm{req}} = \kappa_{\mathrm{cur}}\,\delta_{\mathrm{req}}$.

Unlike in the Lemur software configuration case, IHTML2 also provides the opportunity to change the *structure* of a Web page according to the context. There are two additional tags. The first is `iselect`:

```
<iselect>
  <icase vmod="$\kappa_1$"> $text_1$ </icase>
  ...
  <icase vmod="$\kappa_n$"> $text_n$ </icase>
</iselect>
```

(Of course, `versions`'s could also be present.) If the requested context is κ_{req} and an `iselect` construct is encountered, a corresponding versioned object is created from this delta-versioned object, and then only the code of the *bestfit* `icase` is chosen. The second construct, `icollect`, is similar:

```
<icollect>
  <icase vmod="$\kappa_1$"> $text_1$ </icase>
  ...
  <icase vmod="$\kappa_n$"> $text_n$ </icase>
</icollect>
```

(Of course, `version`'s could also be present.) If the requested context is κ_{req} and an `icollect` construct is encountered, a corresponding versioned object is created from this delta-versioned object, and the code of all of the acceptable `icase`'s is chosen, maintaining a stable order.

A single IHTML source file can specify a whole multiversion family of Web pages. This was different from HTML, in which a link could point just to a unique page. Nowadays, most Web programming is done through scripts, but these are still *not* intensional programs: they do manage *some* variance, but not arbitrary dimensionality.

In IHTML, the link points to a *family* of pages, all held in a single source, and the HTML page produced is calculated based on the current context on the fly. The resulting page is a *version of the Web page*, giving hypertext a new *dimension.* This way of constructing the rendered Web page can be understood as having all the components of the page (images, sections of text, local links, plug-ins, scripts) react simultaneously to the change of the context.

For her PhD work, m.c.schraefel [54] used the IHTML infrastructure so that the dimensions did not correspond to technical attributes but, rather, to more subjective parameters that a reader of a novel might be interested in. She created a multidimensional document discussing *Wuthering Heights*, where the dimensions corresponded to level of detail, perspective, bibliography format, which characters were to be examined, and so on.

2.4.4 Discussion

These developments led Bill Wadge, author Plaice and their colleagues to study more carefully the seminal works on hypertext by Vannevar Bush and Ted Nelson. Bush wrote in 1945 [13] about the *Memex*, a machine in which one can read documents at different speeds or with different levels of detail. Nelson, who coined the term *hypertext* in 1965, also invented the terms *stretchtext*, *plytext* and *poptext*, where, depending of the actions of the user, text looks different. By using the standard intensional programming technique that any problem can be solved by adding a new dimension,[4] Nelson's stretchtext, plytext and poptext are all easily implemented.

Nevertheless, further examination of Nelson's writings [60] showed a significant difference between his view of hypertext and the intensional view. Nelson envisages that any reader can take bits and pieces of other documents to produce a hypertext system with two-way links (to ensure strict copyright control). For him, a hypertext family of documents consists solely of juxtaposing extensions. His family of documents is not an intension, but a collection of extensions. So as concluded in [60], "Hypertext, if it is to be meaningful, can only mean *intensional hypertext*."

At the same time, Yannis Stavrakas, Manolis Gergatsoulis and Panos Rondogiannis continued with the approach of intensionalizing markup languages. They developed Multidimensional XML [87], along with an intensionalized version of XSLT. In the long run, their work should be highly relevant, but currently there is no distribution of any software based on their ideas.

2.5 Remote navigation across possible worlds

In the late 1990s, under the direction of Bill Wadge, Paul Swoboda developed a full context-aware sequential programming language, in which every aspect, including data structures, control structures and functions, was context-aware [88]. We summarize the results here.

2.5.1 Introduction

Possible-worlds versioning allowed the creation of structures that were context-dependent, possible-worlds navigation allowed the creation of context-dependent structures and their automatic recreation as one moved from one context to another. The next step was to integrate these ideas directly into a programming language and to add therein context-dependent mechanisms:

- Context-dependent variables, functions and blocks of code.
- Context deltas upon entry of blocks of code.
- Context deltas upon calling functions.

The mechanisms required for adding these features are similar to those described in the previous section. The additional difficulty is that when a bestfit block of code or a function body is to be executed, the system must determine what should be the new running context. It turns out that there are several choices.

2.5.2 Experiment: Intensional Sequential Evaluator

The Intensional Sequential Evaluator (ISE) language developed by Paul Swoboda is a fully versioned scripting imperative language whose syntax was inspired by Perl4, along with the references (pointers) of Perl5. When an ISE program is called, an execution context is initialized either from an environment variable called `VERSION` or from a command line argument `--version=`*version*. Every aspect of the program execution is based on the context: interpretation of variables, functions, control flow, system calls, etc. During execution, the current context can be modified by applying a delta to it.

In ISE, assignments are versioned, as in:

```
$<lgIn:en>createMapValue = "Create Map";
$<lgIn:fr>createMapValue = "Créez la carte";
$<lgIn:es>createMapValue = "Crear el mapa";
$<>createMapValue        = $<lgIn:en>createMapValue;
```

[4] A variation on Butler Lampson's well known dictum that any problem in computer science can be solved with an extra level of indirection.

`createMapValue` is defined in four versions: the English, French and Spanish of the language interface dimension, and the default version, defined to be the same as the English version.

Deltas follow the scoping rules of the program structure, and can be restricted to individual blocks. Hence, in:

$$\texttt{do}\ \delta\ \texttt{\{}B\texttt{\}} \tag{2.11}$$

the delta δ is applied to the current context before entering block B to create a new current context; upon exit, the previous context is restored. Similarly, in:

$$\texttt{while (}C\texttt{)}\ \delta\ \texttt{\{}B\texttt{\}} \tag{2.12}$$

so long as the conditional expression C is true, block B is executed with delta δ. Slightly more complicated is:

$$\begin{array}{l} \texttt{if (}C_0\texttt{)}\ \delta_0\ \texttt{\{}B_0\texttt{\}} \\ \texttt{elif (}C_1\texttt{)}\ \delta_1\ \texttt{\{}B_1\texttt{\}} \\ \ldots \\ \texttt{else}\ \delta_n\ \texttt{\{}B_n\texttt{\}} \end{array} \tag{2.13}$$

If C_i is the first true expression, block B_i is executed with delta δ_i.

The most complicated structure in ISE is the function call. Like variables, functions — but not their types — can be defined in several versions. This means that there are several function definitions, each tagged with a version, and that the function calls are themselves versioned. The form of a function call is:

$$f\ \mathit{modifier}\ \delta_{\text{req}}\ \delta_{\text{exec}}\ (\mathit{args}) \tag{2.14}$$

where the *modifier* is empty, `?`, or `!`. If we suppose that the current context is κ_{cur}, then in all three cases, the bestfit version of the function is chosen: $(\kappa_{\text{best}}, f_{\text{best}})$. The three different modifiers allow one to determine whether the bestfit body of the function, f_{best}, should be evaluated by applying δ_{exec} to:

- the current context κ_{cur} (empty),
- the requested context $\kappa_{\text{req}} = \kappa_{\text{cur}}\,\delta_{\text{req}}$ (`?`), or
- the bestfit context κ_{best} (`!`).

As a scripting language, ISE was well suited for CGI programming. ISE was used in the first attempts to building a mapping server with a Web interface in an intensional environment ISE [55, 56, 57]. ISE introduced, for the first time, contexts as first-class values. Contexts can be assigned, stored, passed as arguments to functions or even to other processes, and so on. Swoboda [88] used the same runtime system to build VMake and iRCS, versioned variants of make and rcs, as well as to reimplement IHTML, so that they all use the same version space.

2.5.3 Experiment: More programming languages

Wadge developed a markup language with troff syntax called Intensional Markup Language [94]. IML macros are used to generate ISE programs, which, given a multidimensional context, will produce the appropriate HTML page. Using IML notation, the source for multidimensional Web pages becomes substantially more compact than using IHTML or raw ISE.

The idea of versioning commonly used programming languages was studied by two undergraduate student groups at UNSW (Australia) under author Plaice's supervision. Ho, Su and Leung [44] developed an approach to versioning `C`. The Linux Process Control Block was adapted so that each process had a version tag, and system calls were created for manipulating this tag. Versioned `C` functions were implemented using function pointers. Balasingham, Ditu and Hudson [7] experimented with versioning of `C++`, Eiffel and Java. With these developments, it became clear that versioning could be applied to all sorts of different programming tools.

2.6 The evolution of the possible worlds

In 2003, in his PhD thesis [89], supervised under Plaice and Wadge, Swoboda created the *æther*. The æther is an active context which, upon being changed, broadcasts the relevant changes to registered participants. We summarize the results here.

2.6.1 Introduction

The previous three sections showed progressively more sophisticated ways in which a single entity could adapt itself to a single context. But what about the context itself? What is its nature?

Originally, the context was viewed simply as a useful mechanism to hold on to user preferences. But there is another, very fruitful view: the context is *immersive*, or, in other words, the possible world is a *medium* permeating the program. When the context is changed, then like the water passing through our cells, every last subcomponent is affected by the change. As Marshall McLuhan said, "The medium is the message."

This *plenist* viewpoint — as opposed to the *atomist* viewpoint implicit in object-oriented programming — becomes most useful when we consider that *several* objects might share the same context, and that all of these objects are themselves capable of adapting to and modifying the context. In this *intensional community*, a term coined by author Plaice and Kropf [52, 72], the context can be used as a means for broadcasting to all of the other objects sharing this context.

With the *æther*, the active, explicit context, the intensional community can be distinguished from a multi-agent system, which assumes that the agents are simply communicating between themselves through a vacuum, in a point-to-point manner. In an intensional community, programs can communicate either directly, through some communication channel, or *indirectly*, via the context. Changing the context or some part thereof is equivalent to a radio broadcast: all those listening hear; those not listening hear nothing.

An æther contains a context and a set of active participants, each registered at some point in the æther's context. When the æther is modified, deltas are set to all the participants.

2.6.2 Experiment: ISE Æthers

The original æther was developed by Swoboda as a networked server to which ISE programs could connect in order to manipulate named contexts. The server being multithreaded, one ISE process could wait for one of these named contexts to be changed. Should another process change that context, then the waiting process was immediately notified of the new value for the context.

This infrastructure was illustrated with a Web page supporting collaborative browsing [76]. An ISE script was written to present some of the best known paintings of the Louvre, the famous French museum. The result was an intensional Web page with five dimensions: text detail level, text language, image size, painting school and painting reference number. This simple interface was much more intuitive to use than the original Louvre Web page upon which it was based. An additional dimension (to follow, with possible values yes or no) was added to allow collaborative browsing. By adding a single line to the page-generation script, and writing a 20-line wrapper ISE script, people browsing this site could choose to follow what someone else was viewing, while maintaining their own personal preferences, forming in practice an intensional community. However, the ISE æther was more a container for contexts, rather than an active entity.

2.6.3 Experiment: libintense

For his PhD thesis, Swoboda developed a complete suite of programming tools to support æthers as active contexts. These tools include: libintense, an industrial-quality body of code in C++ and Java; and libaep, which supports a new AEPD using active æthers and that allows processes to transparently access æthers over a TCP/IP network. In addition, he extended this basic infrastructure in a number of different directions, including a Perl interface and ijsp, an extension to the Java libintense for the building of intensionalized JSP pages served by an Apache Tomcat server.

The æther is a reactive machine containing a context, and the interface is set up as a subclass of context. Therefore, deltas are applied to æthers in the same manner that they are applied to contexts, with the added side effects of *notifying* the participants of the appropriate deltas. Any process can register a *participant* at a specific node. A participant is a piece of code that is executed when the æther's context is modified at that node or below. Upon context change, the participant is executed, being passed a single argument, namely its relevant delta.

At each instant t, an æther contains:

- κ_t, the current context of the æther;
- P_t, the current set of active participants;
- $D_{p,t}$ (for each participant $p \in P_t$), the set of dimensions being listened to by participant p.

At instant $t + 1$, one, and only one, of the participants may undertake an action:

- $p.$`connect`(D):
  - $\kappa_{t+1} = \kappa_t$;
  - if $D = \varnothing$, then $P_{t+1} = P_t - \{p\}$;
  - if $D \neq \varnothing$, then $P_{t+1} = P_t \cup \{p\}$ and $D_{p,t+1} = D$.
- $p.$`apply`(δ):
  - $\kappa_{t+1} = \kappa_t \, \delta$;
  - for each $p \in P_t$, a delta $\delta_{p,t+1} = \delta | D_{p,t}$ is generated, where $\delta | D_{p,t}$ is the restriction of δ to $D_{p,t}$.

2.6.4 Experiment: Anita Conti Mapping Server

As part of her PhD thesis [58], author Mancilla developed the Anita Conti Mapping Server (ACMS), which provides an intensional Web page, itself containing a intensional map, the two varying with separate context. Each context can be manipulated separately, and can be sensitive to an æther, thereby allowing both the Web page and the map being shown to be changed through the actions of another user, or, conversely, local changes can be broadcast to others. As a result, both the interface and the content of the Web page can be shared with other users.

The ACMS software is interesting because of the nature of its *content. A map* is intrinsically and naturally a multidimensional entity because it depends on a large set of parameters or dimensions of many kinds. We consider an electronic map to be an intension, and that the specific maps generated on demand are the extensions. However, we believe that intensional maps are in some sense more complex than the intensional documents described before, simply because a map is a *visual presentation* of a context — part of the globe's surface — along with certain features linked to that space. Unlike text or software, maps are not representing a sequence of discrete entities like words, characters or glyphs, but, rather, a continuous physical area: a map is not a graph. Magnification of the image should bring in new data from new sources, as appropriate, and in so doing, influence the production of the map, its coloring, contents, frame, titles and labels.

The ACMS was the first full-fledged piece of software that included the `libintense` libraries. Demonstrations given in many different settings were very well received. Nevertheless, the server could not scale, not because of lack of resources but, rather, because there was no means for defining the aggregate semantics of an æther and a set of registered participants.

2.7 Future research: Synchronized possible worlds

The problem outlined in the previous section essentially comes down to unclear timing issues. When there are multiple components sharing the same permeating context, and the components can—in addition to communicating indirectly through the context—communicate directly among themselves, the possibilities for livelocks, deadlocks and other such undesirable phenomena is enormous.

One possibility is to give a completely synchronous semantics to the interaction between æther and participants. Each "instant" would be broken into two:

1. The æther would receive a delta, possibly empty, from each participant, and would then merge these deltas into a single delta, which would then be applied to its context. Should there be inconsistent delta for certain dimensions, then some resolution mechanism would have to be applied, possibly an error, a no-op, or a combination of the different deltas. The æther would then send the relevant deltas to all of the participants.

2. The participants would receive their respective deltas, adapt accordingly, and then proceed to undertake their activity, possibly communicating directly among themselves. Once this activity is finished, then each would send its delta to the æther.

This idea could be extended so that several communities could be hooked up together, all using the same rhythm of alternating æthers/components in action. The intuition is simple, but the details are not: certain dimensions might be controlling critical resources, and would require an appropriate protocol. Similarly, clash resolution needs to be addressed.

2.8 Conclusions

This special issue [of *Mathematics in Computer Science*] focuses on the work undertaken by Bill Wadge or by people he has influenced in topics as diverse as descriptive set theory, logic programming, programming language semantics, hybrid logics, game theory, and other formal work. But Bill's work has always been guided by real, practical problems. In this paper, we have focused on his interest for providing a possible-world semantics for software configuration management, and the surprisingly rich and unexpected consequences of this — on the surface — mundane research topic.

When the original possible-worlds versioning article was submitted to the *IEEE Transactions on Software Engineering*, two of the reviewers made sarcastic comments that Plaice and Wadge wished to do *science fiction*, in order to have spaceships move between possible worlds! In hindsight, we can see that the science-fiction vision of possible worlds has been key in the creation of very useful software.

It is precisely with visions of future possible systems that the ideas described in this paper have been developed. For example, the original article on possible-worlds semantics put forward the idea of a *Montagunix*, a fully versioned Unix-like system; although this idea is now partially available with the development of various flexible Linux distributions, the building process of the latter is clearly not as flexible as the vision of Montagunix.

Similarly, the idea of the intensional communities was created by Plaice and Kropf when they were working on the idea of the *Web Operating System* (WOS), a distributed operating system where the behavior of every component, and their interaction, was fully versioned.

We believe that with the rise of context-aware computing, mobile computing, pervasive computing and ubiquitous computing, that the relevance of possible-worlds versioning will be increasingly felt. We need simple but powerful techniques for adaptation of both the structure and behavior of software.

Part II

Building the Cartesian Space

Chapter 3

Introduction to Core TransLucid

We introduce here the Core TransLucid language through a series of simple examples. In Cartesian programming, as with the Cartesian coordinate system, the key is multidimensionality. For coordinates, a point in one-dimensional space becomes a line in two-dimensional space, a plane in three-dimensional space, a three-dimensional space in four-dimensional space, and so on. Similarly, in Cartesian programming, any entity is considered to "vary" in all dimensions, although this "variance" may well be constant in most dimensions.

The presentation in this chapter begins with a discussion of the Cartesian coordinate system, by focusing on multidimensionality. We introduce the *multidimensional tuple*, which is used to index points, lines, and other structures in this space. We then move on to TransLucid and show how using this tuple and modifying parts thereof can be used to navigate through the space and provoke computation.

The TransLucid primitives are presented informally through the following examples: the factorial, Fibonacci and Ackermann functions; encoding unlimited register machine programs; and triangular matrices. These well-understood examples suffice for presenting the basic language. The rest of the presentation focuses on the two kinds of functional abstraction—value-parameter, using call-by-value, and named-parameter, using call-by-name—and on the use of calculated values as dimensions.

3.1 Cartesian coordinates (two dimensions)

When René Descartes introduced his coordinate system in his *Géométrie*, he used two dimensions, and this is still the way that most people are introduced to this concept in school. For example, in Figure 3.1 we see a discrete two-dimensional grid with x as the abcissa and y as the ordinate.

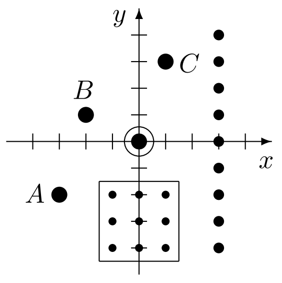


Figure 3.1: A two-dimensional grid in x and y.

Any given point $p = (p_x, p_y)$ in this two-dimensional grid will be written in the TransLucid tuple notation as $[x : p_x, y : p_y]$. Any given horizontal line $y = c_y$ will be written as $[y : c_y]$

while any given vertical line $x = c_x$ will be written as $[x : c_x]$. These tuples can be considered to be indexing a subspace; should two dimensions be defined, a point is designated; should one dimension be defined, a line is designated; the empty tuple [ ] designates the whole space.

The coordinates of the points illustrated in this grid are given below.

- The origin O (◉) has coordinates $[x : 0, y : 0]$.
- Isolated point A (●) has coordinates $[x : -3, y : -2]$.
- Isolated point B (●) has coordinates $[x : -2, y : 1]$.
- Isolated point C (●) has coordinates $[x : 1, y : 3]$.
- The nine points (•) in a vertical line are together designated by coordinate $[x : 3]$.

In addition to points, we can define *intervals* (in one dimension) and *regions* (in two dimensions). The nine points (•) enclosed in a square could be described by the coordinates $[x : -1..1, y : -4..-2]$, where the notation $a..b$, with $a, b \in \mathbb{Z}$ and $a \leqslant b$, means $\{a, a+1, \ldots, b-1, b\}$. However, this new coordinate notation has two possible meanings: Does it mean that $x \in -1..1$ and $y \in -4.. - 2$, i.e., that (x, y) is *one* of those points, or does it mean that $x \equiv -1..1$ and $y \equiv -4.. - 2$, i.e., that (x, y) is the *entire* set of nine points?

In TransLucid, we favor the first interpretation. Equations can be guarded by tuples like $[x : -1..1, y : -4.. - 2]$; the guard is considered to be validated if the current context defines a point within this region.

The second interpretation could be valid, if we allowed set values. Then, inside an expression, $[x : -1..1, y : -4.. - 2]$ would create a tuple where both x and y would take on sets as values. Currently, TransLucid does not support this kind of expression.

3.2 Cartesian coordinates (n dimensions)

Figure 3.2 presents a discrete four-dimensional grid, with dimensions w, z, x and y.

The grid is arranged as a two-dimensional grid with w as the abcissa and z as the ordinate, where every (w, z) point is in fact a two-dimensional grid with x as the abcissa and y as the ordinate. This approach is commonly used to play 4-dimensional tic-tac-toe on a piece of paper. It should be understood that the choice of (x, y) pairs inside (w, z) pairs is completely arbitrary and is simply for visualization purposes on a 2-dimensional visual substrate. In fact, all dimensions are equal, and the order in which they are referred to is irrelevant.

The coordinates of the points illustrated in this grid are given below.

- The origin O (◉) has coordinates $[w : 0, z : 0, x : 0, y : 0]$.
- Isolated point A (●) has coordinates $[w : -2, z : -2, x : -2, y : -2]$.
- Isolated point B (●) has coordinates $[w : -1, z : 1, x : -2, y : 1]$.
- Isolated point C (●) has coordinates $[w : 0, z : 2, x : 0, y : 2]$. The isolated points are all zero-dimensional objects in four-dimensional space because each dimension has been fixed and there is no more variance.
- The five points (●) in a vertical line correspond to $[w : 1, x : 1, y : 1]$. This is a one-dimensional object in four-dimensional space, varying in dimension z.
- The five points (•) in a horizontal line correspond to $[w : -2, z : 0, y : 1]$. This is also a one-dimensional object in four-dimensional space, varying in dimension x.
- The 25 points (•) forming a square correspond to $[w : -1, z : -1]$. This is a two-dimensional object in four-dimensional space, varying in dimensions x and y.

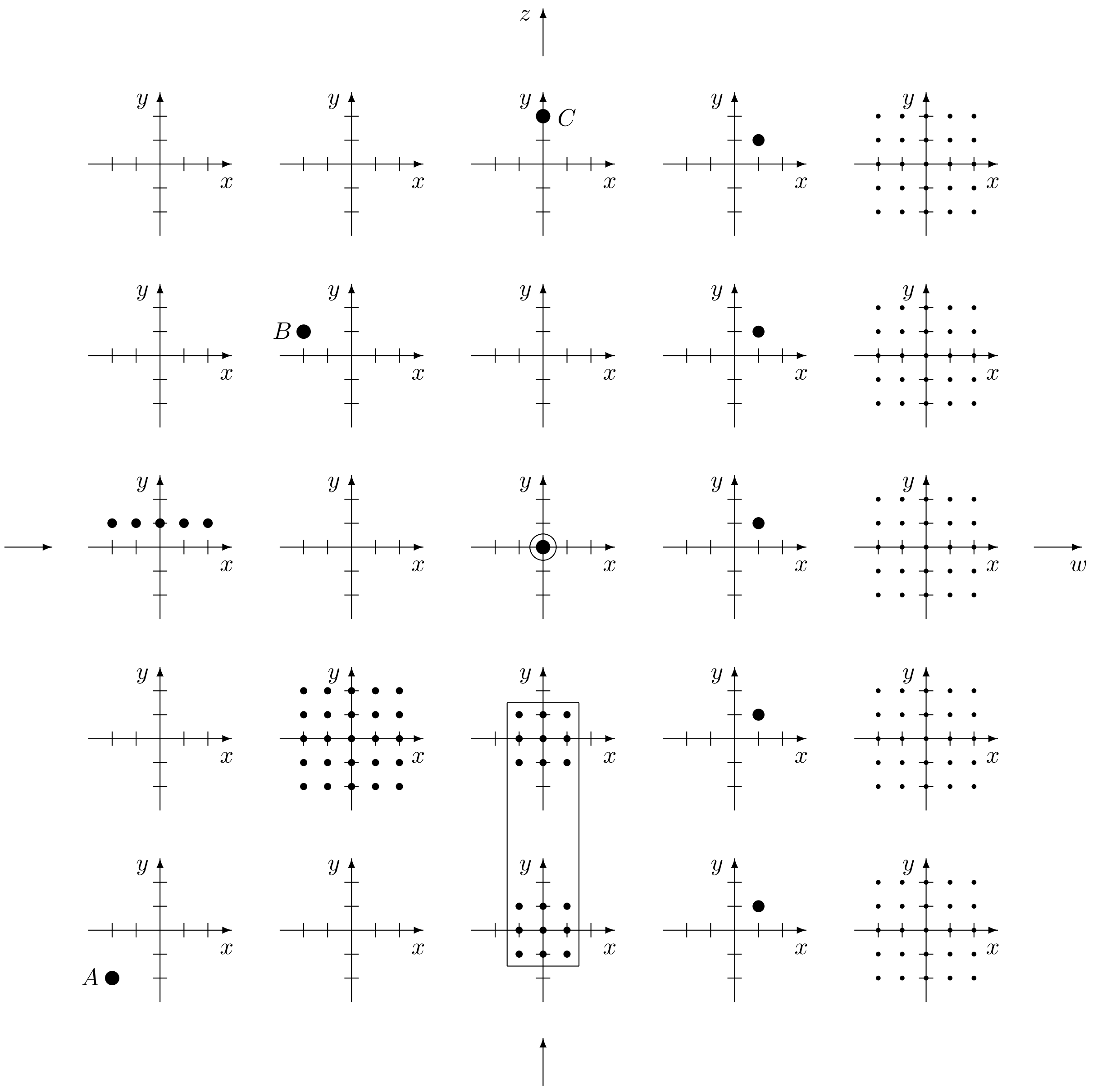


Figure 3.2: A four-dimensional grid in w, z, x and y.

- The 125 points ($\cdot$) forming a rectangle correspond to $[w : 2]$. This is a three-dimensional object in four-dimensional space (only one dimension has been specified), varying in dimensions x, y and z.
- The index [ ] corresponds to the whole space, in this case four-dimensional.

In addition to the points, there is a region consisting of eighteen points ($\bullet$) enclosed in a rectangle. The coordinates for the region are $[w : 0, z : -2.. - 1, x : -1..1, y : -1..1]$.

If we were to introduce a fifth dimension, say u, then the dimensionality of all of the above objects would increase by 1, since none of the tuples refer to dimension u.

This approach can be easily extended to a countably infinite set of dimensions. The use of a tuple then no longer designates a point in a finite-dimensional space but, rather, should be understood as *restricting* variance in an infinite-dimensional space.

The tuple notation can be used to fix a finite set of dimensions with their own values and then all other dimensions to be a specific value $c_{\texttt{all}}$. For example, $[x : 3, y : 4, \texttt{all} : c_{\texttt{all}}]$ defines dimension x to be value 3, dimension y to be value 4, and all other dimensions to be value $c_{\texttt{all}}$.

In what follows, for the purposes of our discussion, we will only need variance in the natural numbers, as opposed to the integers, and dimensions will be numbers rather than letters. This use of values as dimensions is part of the core TransLucid language; the full language allows the use of identifiers as dimensions.

3.3 The TransLucid constant

In TransLucid, expressions are manipulated in an arbitrary dimensional *context*, which corresponds to an index in the Cartesian coordinate framework. As an expression is evaluated, the context may be *queried*, dimension by dimension, in order to produce an answer. In so doing, other expressions may need to be evaluated in other contexts.

The simplest expression in TransLucid is the *constant*. If we consider the value 42, its value is 42, whatever the context. Below, we show the variance of 42 if we allow dimension 0 to vary:

	dim 0 $\rightarrow$						
42	0	1	2	3	4	5	$\cdots$
	42	42	42	42	42	42	$\cdots$

What this table means is that in context $[0 : 0]$, i.e., where dimension 0 takes on the value of 0, the value of expression '42' is 42. The same holds true for all contexts $[0 : i]$, where $i \in \mathbb{N}$.

Here, we show the variance of 42 if we allow dimension 1 to vary:

42	
dim 1 $\downarrow$ 0	42
1	42
2	42
3	42
4	42
5	42
$\vdots$	$\vdots$

So, in context $[1 : 0]$, i.e., where dimension 1 takes on the value of 0, the value of expression '42' is 42. The same holds true for all contexts $[1 : j]$, where $j \in \mathbb{N}$.

The next example uses two dimensions. In context $[0 : 0, 1 : 0]$, i.e., where both dimensions 0 and 1 take on the value 0, the value of expression '42' is still 42. The same holds true for all contexts $[0 : i, 1 : j]$, where $i, j \in \mathbb{N}$.

	dim 0 $\rightarrow$						
42	0	1	2	3	4	5	$\cdots$
dim 1 $\downarrow$ 0	42	42	42	42	42	42	$\cdots$
1	42	42	42	42	42	42	$\cdots$
2	42	42	42	42	42	42	$\cdots$
3	42	42	42	42	42	42	$\cdots$
4	42	42	42	42	42	42	$\cdots$
5	42	42	42	42	42	42	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The 'variance' of '42' in further dimensions would still yield, of course, the same result.

3.4 The TransLucid dimension query

For it to be possible for the context to affect the result of the evaluation of an expression, the context must be *queried*, using the # operator. Below we show how the value of '#0' varies when we let dimension 0 vary:

	dim 0 →						
#0	0	1	2	3	4	5	⋯
	0	1	2	3	4	5	⋯

In, say, context $[0 : 4]$, the value of expression '#0' is 4. In fact, for all contexts $[0 : i]$, where $i \in \mathbb{N}$, the value of #0 is i.

Below, we see the variance of '#0' in two dimensions. There we can see that no matter what the value of dimension 1, only the value of dimension 0 in the current context is of relevance for evaluating expression '#0'. For all contexts $[0 : i, 1 : j]$, where $i, j \in \mathbb{N}$, the value of #0 is i.

	dim 0 →						
#0	0	1	2	3	4	5	⋯
dim 1 ↓ 0	0	1	2	3	4	5	⋯
1	0	1	2	3	4	5	⋯
2	0	1	2	3	4	5	⋯
3	0	1	2	3	4	5	⋯
4	0	1	2	3	4	5	⋯
5	0	1	2	3	4	5	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Below we show how the value of '#1' varies when we let dimension 1 vary:

#1	
dim 1 ↓ 0	0
1	1
2	2
3	3
4	4
5	5
⋮	⋮

In, say, context $[1 : 3]$, the value of expression '#1' is 3. In fact, for all contexts $[1 : j]$, where $j \in \mathbb{N}$, the value of #1 is j.

Below, we see the variance of '#1' in two dimensions. There we can see that no matter what the value of dimension 0, only the value of dimension 1 in the current context is of relevance for evaluating expression '#1'. For all contexts $[0 : i, 1 : j]$, where $i, j \in \mathbb{N}$, the value of #1 is j.

	dim 0 →						
#1	0	1	2	3	4	5	⋯
dim 1 ↓ 0	0	0	0	0	0	0	⋯
1	1	1	1	1	1	1	⋯
2	2	2	2	2	2	2	⋯
3	3	3	3	3	3	3	⋯
4	4	4	4	4	4	4	⋯
5	5	5	5	5	5	5	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

3.5 TransLucid pointwise operators

As in all languages, TransLucid expressions allow the use of operators such as + for addition, $*$ for multiplication, and so on. In TransLucid, these operators are applied *pointwise* to their arguments. In other words, in a given context κ, the expression $E_1 + E_2$ yields the sum of E_1 in κ and of E_2 in κ. Below is the variance of `#0` + `#1` in two dimensions:

	dim 0 →						
`#0` + `#1`	0	1	2	3	4	5	⋯
dim 1 ↓ 0	0	1	2	3	4	5	⋯
1	1	2	3	4	5	6	⋯
2	2	3	4	5	6	7	⋯
3	3	4	5	6	7	8	⋯
4	4	5	6	7	8	9	⋯
5	5	6	7	8	9	10	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

In context $[0 : i, 1 : j]$, for all $i, j \in \mathbb{N}$, `#0` + `#1` has the value $i + j$.

There is also in TransLucid a conditional expression, `if–then–else`, whose meaning is also given pointwise. The difference between the conditional and, say, +, is that the conditional is not strict in its last two arguments: only one of them will be evaluated, depending on the value of the condition.

	dim 0 →						
`if #0` % 2 ≡ 0 `then #1 else` − `#1`	0	1	2	3	4	5	⋯
dim 1 ↓ 0	0	0	0	0	0	0	⋯
1	1	−1	1	−1	1	−1	⋯
2	2	−2	2	−2	2	−2	⋯
3	3	−3	3	−3	3	−3	⋯
4	4	−4	4	−4	4	−4	⋯
5	5	−5	5	−5	5	−5	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

3.6 Change of context

If the context can be queried in TransLucid, then it also needs to be changeable. This is done using the `@` operator, which takes a tuple as parameter and uses it to change the current context before continuing with the evaluation of the expression. Below, the expression `#0` + `#1` is evaluated in a new context, which is created by incrementing the 0 dimension's ordinate by 1.

	dim 0 →						
(`#0` + `#1`) `@` [0 : `#0` + 1]	0	1	2	3	4	5	⋯
dim 1 ↓ 0	1	2	3	4	5	6	⋯
1	2	3	4	5	6	7	⋯
2	3	4	5	6	7	8	⋯
3	4	5	6	7	8	9	⋯
4	5	6	7	8	9	10	⋯
5	6	7	8	9	10	11	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

As a result, in context $[0 : i, 1 : j]$, expression (`#0` + `#1`) `@` [0 : `#0` + 1] evaluates to $i + j + 1$.

3.7 Factorial function in TransLucid

TransLucid of course needs *variables* and equations to define these. We introduce these with the factorial function, whose first few entries can be found below:

	dim 0 $\rightarrow$											
fact	0	1	2	3	4	5	6	7	8	9	10	···
	1	1	2	6	24	120	720	5040	40320	362880	3628800	···

The definition in TransLucid is recursive, with a base case and an inductive case.

$$\mathit{fact} = \texttt{if}\ \texttt{\#}0 \equiv 0\ \texttt{then}\ 1\ \texttt{else}\ \mathit{fact}\ \texttt{@}\ [0 : \texttt{\#}0 - 1] * \texttt{\#}0$$

To evaluate 5!, we must ask for it. Equations do not run, they simply are. The demand for 5! yields the following linear expansion:

$$\begin{aligned}
&\mathit{fact}\ \texttt{@}\ [0:5] \\
\rightarrow\ &\mathit{fact}\ \texttt{@}\ [0:4] * 5 \\
\rightarrow\ &\mathit{fact}\ \texttt{@}\ [0:3] * 4 * 5 \\
\rightarrow\ &\mathit{fact}\ \texttt{@}\ [0:2] * 3 * 4 * 5 \\
\rightarrow\ &\mathit{fact}\ \texttt{@}\ [0:1] * 2 * 3 * 4 * 5 \\
\rightarrow\ &\mathit{fact}\ \texttt{@}\ [0:0] * 1 * 2 * 3 * 4 * 5 \\
\rightarrow\ &1 * 1 * 2 * 3 * 4 * 5 \\
\rightarrow\ &120
\end{aligned}$$

3.8 Bestfitting of definitions

The factorial function can be rewritten with a pair of definitions, where the current context is used to choose which is most appropriate. This choice is called *bestfitting.* The way that it works is that each definition q is valid for a certain region K_q, called the *context guard.* Given a current context κ, the *applicable* definitions are the ones for which $\kappa \in K_q$. The *bestfit* definition q_{best} is the one for which $K_{q_{\text{best}}} \subseteq K_q$ for all applicable q.

For factorial, there are two definitions, the $[0 : 0]$ case and the default case.

$$\begin{aligned}
&\mathit{fact} \mid [0:0] = 1 \\
&\mathit{fact} = \mathit{fact}\ \texttt{@}\ [0 : \texttt{\#}0 - 1] * \texttt{\#}0
\end{aligned}$$

3.9 Fibonacci function in TransLucid

The Fibonacci series is probably the best-known series. Its first few entries are:

	dim 0 $\rightarrow$											
fib	0	1	2	3	4	5	6	7	8	9	10	···
	0	1	1	2	3	5	8	13	21	34	55	···

The TransLucid version uses three definitions, two base cases and the iterative case.

$$\begin{aligned}
&\mathit{fib} \mid [0:0] = 0 \\
&\mathit{fib} \mid [0:1] = 1 \\
&\mathit{fib} = \mathit{fib}\ \texttt{@}\ [0 : \texttt{\#}0 - 1] + \mathit{fib}\ \texttt{@}\ [0 : \texttt{\#}0 - 2]
\end{aligned}$$

The naïve evaluation provokes a branching-style recursion with much repetition.

$$\begin{aligned}
& \mathit{fib}\ \texttt{@}\ [0:4] \\
\to\ & \mathit{fib}\ \texttt{@}\ [0:3] + \mathit{fib}\ \texttt{@}\ [0:2] \\
\to\ & \mathit{fib}\ \texttt{@}\ [0:2] + \mathit{fib}\ \texttt{@}\ [0:1] + \mathit{fib}\ \texttt{@}\ [0:1] + \mathit{fib}\ \texttt{@}\ [0:0] \\
\to\ & \mathit{fib}\ \texttt{@}\ [0:1] + \mathit{fib}\ \texttt{@}\ [0:0] + 1 + 1 + 0 \\
\to\ & 1 + 0 + 1 + 1 + 0 \\
\to\ & 3
\end{aligned}$$

3.10 Ackermann function in TransLucid

The Ackermann function is one of the first recursive functions discovered that is not primitive recursive. It grows so fast that in general it cannot be computed once its first argument is greater than 3.

	dim 0 →									
ack	0	1	2	3	4	5	6	7	8	⋯
dim 1 ↓ 0	1	2	3	4	5	6	7	8	9	⋯
1	2	3	4	5	6	7	8	9	10	⋯
2	3	5	7	9	11	13	15	17	19	⋯
3	5	13	29	61	125	253	509	1021	2045	⋯
4	13	65533	⋯							
5	65533	⋯								
⋮	⋱									

Ackermann in TransLucid uses four definitions, to ensure that there is no ambiguity with the bestfitting process.

$$\begin{aligned}
& \mathit{ack} \mid [1:0, 0:0] = 1 \\
& \mathit{ack} \mid [1:0] = \texttt{\#0} + 1 \\
& \mathit{ack} \mid [0:0] = \mathit{ack}\ \texttt{@}\ [1:\texttt{\#1} - 1, 0:1] \\
& \mathit{ack} = \mathit{ack}\ \texttt{@}\ \big[1:\texttt{\#1} - 1, 0:\mathit{ack}\ \texttt{@}\ [0:\texttt{\#0} - 1]\big]
\end{aligned}$$

In the iterative case, note that the nested context change is only changing the value for dimension 0, since the value for dimension 1 need not be changed. This is similar to what happens with differential equations, in which only the definitions of relevance are written down.

The naïve evaluation of Ackermann takes place with a lot of nested recursive calls.

$$\begin{aligned}
&\ \mathit{ack}\ \texttt{@}\ [1:3,\ 0:0] \\
&\to \mathit{ack}\ \texttt{@}\ [1:2,\ 0:1] \\
&\to \mathit{ack}\ \texttt{@}\ \big[1:1,\ 0:\mathit{ack}\ \texttt{@}\ [1:2,\ 0:0]\big] \\
&\to \mathit{ack}\ \texttt{@}\ \big[1:1,\ 0:\mathit{ack}\ \texttt{@}\ [1:1,\ 0:1]\big] \\
&\to \mathit{ack}\ \texttt{@}\ \Big[1:1,\ 0:\mathit{ack}\ \texttt{@}\ \big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ [1:1,\ 0:0]\big]\Big] \\
&\to \mathit{ack}\ \texttt{@}\ \Big[1:1,\ 0:\mathit{ack}\ \texttt{@}\ \big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ [1:0,\ 0:1]\big]\Big] \\
&\to \mathit{ack}\ \texttt{@}\ \big[1:1,\ 0:\mathit{ack}\ \texttt{@}\ [1:0,\ 0:2]\big] \\
&\to \mathit{ack}\ \texttt{@}\ [1:1,\ 0:3] \\
&\to \mathit{ack}\ \texttt{@}\ \big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ [1:1,\ 0:2]\big] \\
&\to \mathit{ack}\ \texttt{@}\ \Big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ \big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ [1:1,\ 0:1]\big]\Big] \\
&\to \mathit{ack}\ \texttt{@}\ \Bigg[1:0,\ 0:\mathit{ack}\ \texttt{@}\ \Big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ \big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ [1:1,\ 0:0]\big]\Big]\Bigg] \\
&\to \mathit{ack}\ \texttt{@}\ \Bigg[1:0,\ 0:\mathit{ack}\ \texttt{@}\ \Big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ \big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ [1:0,\ 0:1]\big]\Big]\Bigg] \\
&\to \mathit{ack}\ \texttt{@}\ \Big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ \big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ [1:0,\ 0:2]\big]\Big] \\
&\to \mathit{ack}\ \texttt{@}\ \big[1:0,\ 0:\mathit{ack}\ \texttt{@}\ [1:0,\ 0:3]\big] \\
&\to \mathit{ack}\ \texttt{@}\ [1:0,\ 0:4] \\
&\to 5
\end{aligned}$$

3.11 Unlimited register machines

One of the best known models for computation is the *unlimited register machine* [21]. In this machine, there is an infinite set of registers R_i, $i > 0$, most of which are initialized to the value 0. The n arguments to a program are placed in registers $R_1, \ldots, R_n$. A program is made of instructions numbered from 1 to m. The program is started on instruction 1 and terminates with the result in R_1 when the program counter is greater than m. The possible instructions are:

$\texttt{Z}(i)$	Zero R_i, go to next instruction.
$\texttt{S}(i)$	Increment R_i by 1, go to next instruction.
$\texttt{T}(i,j)$	Copy the contents of R_i into R_j, go to next instruction.
$\texttt{J}(i,j,k)$	If the contents of R_i and R_j are equal, go to instruction k, otherwise the next instruction.

For example, the five instructions below compute $R_1 - R_2$, assuming that $R_1 \geqslant R_2$.

$$\begin{aligned}
&1:\texttt{J}(1,2,5)\\
&2:\texttt{S}(2)\\
&3:\texttt{S}(3)\\
&4:\texttt{J}(1,1,1)\\
&5:\texttt{T}(3,1)
\end{aligned}$$

Programs written for an unlimited register machine of length m can be encoded directly in TransLucid using $m + 1$ definitions. The translation of a program $I_1, \ldots, I_m$ is as follows, where

$\mathcal{T}$ is the transformer of an instruction.

$$\begin{aligned}
&X \mid [0:1] = \mathcal{T}(I_1)\\
&\dots\\
&X \mid [0:m] = \mathcal{T}(I_m)\\
&X = \texttt{\#}1
\end{aligned}$$

Here is the definition of $\mathcal{T}$. Dimension 0 encodes the program counter.

$$\begin{aligned}
&\mathcal{T}\big(\texttt{Z}(i)\big) = X \texttt{ @ } [0:\texttt{\#}0+1, i:0]\\
&\mathcal{T}\big(\texttt{S}(i)\big) = X \texttt{ @ } [0:\texttt{\#}0+1, i:\texttt{\#}i+1]\\
&\mathcal{T}\big(\texttt{T}(i,j)\big) = X \texttt{ @ } [0:\texttt{\#}0+1, j:\texttt{\#}i]\\
&\mathcal{T}\big(\texttt{J}(i,j,k)\big) = \texttt{if } \texttt{\#}i \equiv \texttt{\#}j \texttt{ then } X \texttt{ @ } [0:k] \texttt{ else } X \texttt{ @ } [0:\texttt{\#}0+1]
\end{aligned}$$

Here is the translation of the subtraction program.

$$\begin{aligned}
&\mathit{sub} \mid [0:1] = \texttt{if } \texttt{\#}1 \equiv \texttt{\#}2 \texttt{ then } \mathit{sub} \texttt{ @ } [0:5] \texttt{ else } \mathit{sub} \texttt{ @ } [0:\texttt{\#}0+1]\\
&\mathit{sub} \mid [0:2] = \mathit{sub} \texttt{ @ } [0:\texttt{\#}0+1, 2:\texttt{\#}2+1]\\
&\mathit{sub} \mid [0:3] = \mathit{sub} \texttt{ @ } [0:\texttt{\#}0+1, 3:\texttt{\#}3+1]\\
&\mathit{sub} \mid [0:4] = \texttt{if } \texttt{\#}1 \equiv \texttt{\#}1 \texttt{ then } \mathit{sub} \texttt{ @ } [0:1] \texttt{ else } \mathit{sub} \texttt{ @ } [0:\texttt{\#}0+1]\\
&\mathit{sub} \mid [0:5] = \mathit{sub} \texttt{ @ } [0:\texttt{\#}0+1, 1:\texttt{\#}3]\\
&\mathit{sub} = \texttt{\#}1
\end{aligned}$$

The demand for the computation of $4-2$ requires the use of the `all` notation for tuples, to indicate that the default initial value for all of the registers is 0.

$$\begin{aligned}
&\ \mathit{sub} \texttt{ @ } [0:1, 1:4, 2:2, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:2, 1:4, 2:2, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:3, 1:4, 2:3, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:4, 1:4, 2:3, 3:1, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:1, 1:4, 2:3, 3:1, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:2, 1:4, 2:3, 3:1, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:3, 1:4, 2:4, 3:1, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:4, 1:4, 2:4, 3:2, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:1, 1:4, 2:4, 3:2, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:5, 1:4, 2:4, 3:2, \texttt{all}:0]\\
&\rightarrow \mathit{sub} \texttt{ @ } [0:6, 1:2, 2:4, 3:2, \texttt{all}:0]\\
&\rightarrow 2
\end{aligned}$$

3.12 Boolean guards

It is sometimes the case that the context guard is not sufficient to restrict the applicability of a definition. To do this, we introduce as well the *Boolean guard*. For example, the first definition given below is only valid if the current context is included in $[0:0..7, 1:0..7]$, and if $\texttt{\#}0 \geqslant \texttt{\#}1$.

$$\begin{aligned}
&\mathit{triangle} \mid [0:0..7, 1:0..7]\ \&\ \texttt{\#}0 \geqslant \texttt{\#}1 = \texttt{\#}0 + \texttt{\#}1\\
&\mathit{triangle} = 0
\end{aligned}$$

The result is to define an upper-triangular matrix.

	dim 0 →								
triangle	0	1	2	3	4	5	6	7	⋯
dim 1 ↓ 0	0	1	2	3	4	5	6	7	⋯
1	0	2	3	4	5	6	7	8	⋯
2	0	0	4	5	6	7	8	9	⋯
3	0	0	0	6	7	8	9	10	⋯
4	0	0	0	0	8	9	10	11	⋯
5	0	0	0	0	0	10	11	12	⋯
6	0	0	0	0	0	0	12	13	⋯
7	0	0	0	0	0	0	0	14	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

3.13 Functional abstraction

Adding functions and `where` clauses to TransLucid is based on the ideas used in Indexical Lucid [37]. A function can carry both *value parameters* and *named parameters*. Standard examples are:

$$\begin{aligned}
\texttt{index}.d &= \texttt{\#}d + 1 \;;; \\
\texttt{first}.d\, X &= X \;\texttt{@}\; [d : 0] \;;; \\
\texttt{prev}.d\, X &= X \;\texttt{@}\; [d : \texttt{\#}d - 1] \;;; \\
\texttt{next}.d\, X &= X \;\texttt{@}\; [d : \texttt{\#}d + 1] \;;; \\
X\ \texttt{fby}.d\, Y &= \texttt{if}\ \texttt{\#}d \leqslant 0\ \texttt{then}\ X\ \texttt{else}\ \texttt{prev}.d\, Y\ \texttt{fi} \;;; \\
X\ \texttt{before}.d\, Y &= \texttt{if}\ \texttt{\#}d \geqslant 0\ \texttt{then}\ Y\ \texttt{else}\ \texttt{next}.d\, X\ \texttt{fi} \;;;
\end{aligned}$$

In each of these examples, each of the declared functions has a single dimensional parameter d. `index` takes no named parameters, `first`, `prev`, and `next` take one named parameter each, while `fby` and `before` take two named parameters each.

The idea of the value parameters is that they are used to specify either constant values or the dimensions of relevance in the hyperdatons being passed. For this to work, the value parameters need to be evaluated prior to the function being evaluated, so they are passed call-by-value. As for the named parameters, since they are typically infinite data structures, they are passed call-by-name.

The `where` clause allows the declaration of both local dimensions and local variables.

```
E
where
  dimension d_1 , ... , d_m ;;
  x_1=E_1 ;;
  ...
  x_n=E_n ;;
end
```

Dimensions d_1 through d_m are only visible in expressions $E, E_1, \ldots, E_n$, and are all initialized to an initial value, typically 0.

3.14 Simple recursive functions

The basic recursive functions presented earlier in the chapter can now be defined, not as single hyperdatons, but as functions.

```
fact.n = f
where
  f = 1 fby.n f * index.n ;;
end

fib.n = f
where
  f = 0 fby.n 1 fby.n f + next.n f ;;
end

ack.m.n = f
where
  f = index.n fby.m (next.n f fby.n next.m f) ;;
end
```

The m and n value parameters allow the creation of a number of instances of these functions, each varying in a different parameter. For example, *fact*.3 corresponds to:

	dim 3 $\rightarrow$											
fact.3	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
	1	1	2	6	24	120	720	5040	40320	362880	3628800	$\cdots$

while *fact.42* corresponds to:

	dim 42 $\rightarrow$											
fact.42	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
	1	1	2	6	24	120	720	5040	40320	362880	3628800	$\cdots$

3.15 Matrix operations

Matrices naturally vary in two dimensions, but several operations require an additional dimension. For example, to do multiplication, two matrices, A and B, vary in dimensions x and y. To multiply them together, one rotates the y part of A into temporary direction z, and the same for the x part of B, then these two rotated matrices are multiplied pointwise, then summed in the z direction. Here is the multiplication operation, where the n parameter corresponds to the number of elements in the y direction for A and the x direction for B.

```
rotate.x.y A = A @ [x : #y] ;;

sum.d.n X = Y @ [d : n − 1]
where
  Y = X fby.d Y + next.d X ;;
end

multiply.x.y.n A B = D
where
  dimension z ;;
  C = rotate.z.y A * rotate.z.x B ;;
  D = sum.z.n C ;;
end
```

3.16 Dataflow filters

In *Lucid, the Dataflow Programming Language* [95], Wadge and Ashcroft describe a number of dataflow filters, defined as recursively defined functions, with every recursive application advancing on one or more of the arguments. The most important of these filters are `wvr` ("whenever"), `upon` ("advance upon") and `merge` ("merge sort"). We give their definitions assuming that the streams are infinite.

```
X wvr.d Y =
  if first.d Y
  then X fby.d (next.d X wvr.d next.d Y)
  else next.d X wvr.d next.d Y fi ;;

X upon.d Y =
  X fby.d if first.d Y
          then next.d X upon.d next.d Y
          else X upon.d next.d Y fi ;;

X merge.d Y =
  if first.d X < first.d Y
    then X fby.d (next.d X merge.d Y)
  elsif first.d X > first.d Y
    then Y fby.d (X merge.d next.d Y)
  else X fby.d (next.d X merge.d next.d Y) fi ;;
```

A simple example of the use of `wvr` is the Sieve of Erastosthenes for computing the primes:

```
sieve.d = f
where
  dimension z ;;
  f = (#z + 2) fby.d (f wvr.z (f mod first.z f ≠ 0)) ;;
end
```

The sequence of primes is formed by the first element in each row:

		dim z →													
	sieve.d	0	1	2	3	4	5	6	7	8	9	10	11	12	⋯
dim d ↓	0	2	3	4	5	6	7	8	9	10	11	12	13	14	⋯
	1	3	5	7	9	11	13	15	17	19	21	23	25	27	⋯
	2	5	7	11	13	17	19	23	25	29	31	35	37	41	⋯
	3	7	11	13	17	19	23	29	31	37	41	43	47	49	⋯
	4	11	13	17	19	23	29	31	37	41	43	47	53	59	⋯
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Each row corresponds to the removal of the multiples of the first element in the previous row.

A simple example of the use of *merge* is the Hamming problem for computing, in order, with no repetition, all numbers of the form $2^i 3^j 5^k$:

```
hamming.d = h
where
  h = 1 fby.d (2 * h) merge.d (3 * h) merge.d (5 * h) ;;
end
```

Here are the first few values of Hamming:

	dim d $\rightarrow$													
hamming.d	0	1	2	3	4	5	6	7	8	9	10	11	12	$\cdots$
	1	2	3	4	5	6	8	9	10	12	15	16	18	$\cdots$

The *merge* filter can be written using `upon`:

```
$X$ merge.$d$ $Y$ = if $X' \leqslant Y'$ then $X'$ else $Y'$ fi
where
   $X' = X$ upon.$d$ $X' \leqslant Y'$ ;;
   $Y' = Y$ upon.$d$ $X' \geqslant Y'$ ;;
end
```

3.17 Higher-order functions

The syntax used for the functions presented in the previous sections shows that these functions are Curryfied, i.e., they take their arguments one at a time. Hence when we write

$$multiply.x.y.n\ A\ B = D\ \cdots ,$$

multiply is understood to take five arguments, one at a time, and that four intermediate functions are created. In functional programming, this is done by using λ-abstraction. The same is done here, but a distinction must be made between value and named parameters. Therefore we use λ-abstraction for value parameters and ϕ-abstraction for named parameters. So we can write

$$multiply = \lambda x.\ \lambda y.\ \lambda n.\ \phi A.\ \phi B.\ D\ \cdots .$$

If there are two different kinds of abstraction, then two different kinds of application are also needed. λ-application requires a period between function and argument, while ϕ-application simply requires the juxtaposition of function and argument.

In the final section of *Lucid, the Dataflow Programming Language* [95], Wadge and Ashcroft explained that the addition of functions to Lucid was problematic, because Lucid objects are infinite. One would like to apply higher-order functions to infinite objects, to build infinite objects containing higher-order functions, and even to build infinite objects containing higher-order functions that can be applied to infinite objects.

The very last example of the Lucid book posited that one could build streams of functions, using λ-expressions. We go further, by allowing entire *hyperdatons of functions*, i.e., infinite multidimensional families of functions. We simply index all of the occurrences of λ or ϕ. Here is such a family:

```
$powers.x.y = f$
where
   $f = (\lambda s.1)$ fby.$x$ $\big((\lambda s.s * f\ s)$ fby.$y$ $(\lambda s.s * f\ s)\big)$
end
```

It defines the following table of functions:

	dim x $\rightarrow$						
powers.x.y	0	1	2	3	4	5	$\cdots$
dim y $\downarrow$ 0	$\lambda s.s^0$	$\lambda s.s^1$	$\lambda s.s^2$	$\lambda s.s^3$	$\lambda s.s^4$	$\lambda s.s^5$	$\cdots$
1	$\lambda s.s^1$	$\lambda s.s^2$	$\lambda s.s^3$	$\lambda s.s^4$	$\lambda s.s^5$	$\lambda s.s^6$	$\cdots$
2	$\lambda s.s^2$	$\lambda s.s^3$	$\lambda s.s^4$	$\lambda s.s^5$	$\lambda s.s^6$	$\lambda s.s^7$	$\cdots$
3	$\lambda s.s^3$	$\lambda s.s^4$	$\lambda s.s^5$	$\lambda s.s^6$	$\lambda s.s^7$	$\lambda s.s^8$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

3.18 Values as dimensions

The most notable feature of TransLucid with respect to previous languages derived from Lucid is the fact that dimensions are first-class values. Below we present an example from 3D particle-in-cell simulation in which having dimensions as values simplifies the writing of the code. The discussion below is inspired by Tristan, a FORTRAN program for the simulation of plasmas developed by the late Oscar Buneman [70]. and a discussion in Joey Paquet's PhD thesis [66].

If the units are chosen appropriately, the first two equations of Maxwell's equations become [70]:

$$\frac{\partial \mathbf{B}}{\partial t} = -c(\nabla \times \mathbf{E}) \tag{3.1}$$

$$\frac{\partial \mathbf{E}}{\partial t} = c(\nabla \times \mathbf{B}) - \mathbf{j} \tag{3.2}$$

where $\mathbf{B}$ is the magnetic field, $\mathbf{E}$ is the electric field, $\mathbf{j}$ is the electric charge, and the curl is:

$$\nabla \times\ \mathbf{F} = \hat{\mathbf{x}}\left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z}\right) + \hat{\mathbf{y}}\left(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x}\right) + \hat{\mathbf{z}}\left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y}\right)$$

To simulate these equations, space is partitioned into 3-dimensional cells and each cell memorizes the value for each of the components of the local magnetic and electric field : $\mathbf{B}_x$, $\mathbf{B}_y$, $\mathbf{B}_z$, $\mathbf{E}_x$, $\mathbf{E}_y$ and $\mathbf{E}_z$, where $\mathbf{F}_x$ means component x of vector $\mathbf{F}$. However, to ensure reasonable accuracy in the use of difference equations, a leapfrogging technique is used: the values of the components given above do not all correspond to the values at the origin of the cell, but, rather, to the following values. Note that we use **bold** for continuous and `typewriter` for discrete fields.

$$\begin{aligned}
{}_{xyzt}\mathtt{E}_x &= {}_{(x+.5)yz(t+.5)}\mathbf{E}_x \\
{}_{xyzt}\mathtt{E}_y &= {}_{x(y+.5)z(t+.5)}\mathbf{E}_y \\
{}_{xyzt}\mathtt{E}_z &= {}_{xy(z+.5)(t+.5)}\mathbf{E}_z \\
{}_{xyzt}\mathtt{B}_x &= {}_{x(y+.5)(z+.5)t}\mathbf{B}_x \\
{}_{xyzt}\mathtt{B}_y &= {}_{(x+.5)y(z+.5)t}\mathbf{B}_y \\
{}_{xyzt}\mathtt{B}_z &= {}_{(x+.5)(y+.5)zt}\mathbf{B}_z
\end{aligned}$$

To simplify the presentation, we can summarize the above six lines by:

$$\mathtt{E} = {}_{(m+.5)(t+.5)}\mathbf{E}$$
$$\mathtt{B} = {}_{(l+.5)(r+.5)}\mathbf{B}$$

where index m corresponds to the component that is of interest, l to the component to the left and r to the component to the right. With the same notation, the curl becomes:

$$\nabla \times\ \mathbf{F} \quad = \quad \Big({}_{(r+\delta)}F_l - {}_{(r-\delta)}F_l\Big) - \Big({}_{(l+\delta)}F_r - {}_{(l-\delta)}F_r\Big)$$

where δ is an infinitely small value. But this same expression can also be used to define finite difference equations, by allowing the δ to become a standard real value.

Consider Equation 3.1.

$$\frac{\partial \mathbf{B}}{\partial t} = -c(\nabla \times \mathbf{E})$$

This equation can be rewritten, using an infinitesimal δ, as:

$$\begin{aligned}
&\frac{{}_{(t+\delta)}\mathbf{B} - {}_{(t-\delta)}\mathbf{B}}{2\delta} \\
= &-c\Big(\big({}_{(r+\delta)}\mathbf{E}_l - {}_{(r-\delta)}\mathbf{E}_l\big) - \big({}_{(l+\delta)}\mathbf{E}_r - {}_{(l-\delta)}\mathbf{E}_r\big)\Big)
\end{aligned}$$

If we shift by $(t+\delta)(l+\delta)(r+\delta)$, we get:

$$\frac{{}_{(t+2\delta)(l+\delta)(r+\delta)}\mathbf{B} - {}_{(l+\delta)(r+\delta)}\mathbf{B}}{2\delta}$$
$$= -c\Big(\big({}_{(t+\delta)(l+\delta)(r+2\delta)}\mathbf{E}_l - {}_{(t+\delta)(l+\delta)}\mathbf{E}_l\big) - \big({}_{(t+\delta)(l+2\delta)(r+\delta)}\mathbf{E}_r - {}_{(t+\delta)(r+\delta)}\mathbf{E}_r\big)\Big)$$

Now if we let $\delta = .5$, we get:

$$_{(t+1)}\mathtt{B} - \mathtt{B} = -c\Big(\big({}_{(r+1)}\mathtt{E}_l - \mathtt{E}_l\big) - \big({}_{(l+1)}\mathtt{E}_r - \mathtt{E}_r\big)\Big) \tag{3.3}$$

which can now be encoded using TransLucid. Similar reasoning gives:

$$_{(t+1)}\mathtt{E} - \mathtt{E} = c\Big(\big({}_{(t+1)}\mathtt{B}_l - {}_{(t+1)(r-1)}\mathtt{B}_l\big) - \big({}_{(t+1)}\mathtt{B}_r - {}_{(t+1)(l-1)}\mathtt{B}_r\big)\Big) - {}_{(t+1)(m+.5)}\,\mathtt{q} \tag{3.4}$$

The implementation in TransLucid uses five dimensions:

- dimensions 0, 1 and 2, respectively, for x, y and z;
- dimension `dim`, which takes on values 0, 1 and 2;
- dimension `t`, for time.

In other words, at each (x, y, z, t) point, there is an x-value, a y-value and a z-value.

To compute the curl, we need the following three functions.

```
dprev.d X = X − prev.d X;

dnext.d X = next.d X − X;

curl.diff F =
  diff.r (F @ [dim : ℓ]) − diff.ℓ (F @ [dim : r])
where
  r = (#dim + 1) mod 3 ;;
  ℓ = (#dim + 2) mod 3 ;;
end
```

Now, the curl in Equation 3.3, for the magnetic field, becomes:

$$\mathit{curl.dnext}\ E$$

while the curl in Equation 3.4, for the electric field, becomes:

$$\mathit{curl.dprev}\ (\mathtt{next}.t\ B)$$

By allowing the space dimensions to be defined by the values 0, 1 and 2, the definition of the curl becomes very simple.

3.19 Conclusions

To understand Core TransLucid does not require any sophisticated mathematics, simply high-school mathematics, specifically the multidimensional coordinate system.

A Core TransLucid program consists of a set of guarded equations and a set of expressions to be evaluated. "Running" the program consists in evaluating the expressions. Initially, an expression is evaluated with respect to an empty current context; as it gets evaluated, the `@` operator is used to change the current context, and the `#` operator is used to query the current context. When an identifier x is encountered in an expression, the bestfit definition for x must be sought out.

The guard in an equation has two components, the context guard and the Boolean guard. Either of these may be empty. For the equation to be applicable, the current context must be contained in the context region defined by the context guard, and, if this is the case, the Boolean guard must evaluate to true in the current context. Once the applicable equations have been selected, then the bestfit equation is chosen by selecting the one whose context guard defines a region that is wholy contained in all of the others.

In the rest of the thesis, we will be examining many different aspects of computer science. Nevertheless, the key concepts of the thesis have all been presented here: everything will ultimately be understood in terms of context change, context query and bestfitting.

Chapter 4

The Syntax and Semantics of Core TransLucid

In this chapter, we formally present the Core TransLucid. We begin with the abstract syntax and the denotational semantics, using an environment that maps identifiers to *intensions*, which map contexts to values. We then introduce an operational semantics that corresponds structurally to the denotational semantics, by making demands for (*identifier*, *context*) pairs, which in turn may provoke demands for other such pairs, and demonstrate the equivalence of the two semantics for producing correct values. Function calls are transformed into changes of context, through the use of an index, used as a dimension, to keep track of the different occurrences of λ and ϕ; for value-parameter application, this dimension takes on the value of the argument, while for named-value application, this dimension takes on a list keeping track of all of the arguments (expressions) that have been called in previous instantiations. We then show how the operational semantics can be adapted to include caching of previously computed values, where the cache resembles the environment of the denotational semantics, but ensures that only dimensions of relevance to the computation are stored in the cache.

4.1 Syntax

A Core TransLucid program consists of a set of equations followed by an expression to be evaluated.

$$
\begin{aligned}
E ::= {} & x \\
\mid {} & c \\
\mid {} & op(E_1, \ldots, E_n) \\
\mid {} & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ fi} \\
\mid {} & \texttt{\#}E \\
\mid {} & [E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2}] \\
\mid {} & [E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2},\ \texttt{all} : E_{\texttt{all}}] \\
\mid {} & E_2 \texttt{ @ } E_1 \\
\mid {} & (\lambda x.E) \\
\mid {} & E_1.E_2 \\
\mid {} & (\phi x.E) \\
\mid {} & E_1\ E_2 \\
\mid {} & E \texttt{ where } d_{i=1..m}\ q_{j=1..n}
\end{aligned}
$$

$$q ::= x_q = g_{i=1..n}$$

$$g ::= B_g \to E_g$$

where

- E is an expression;
- x is an identifier;
- c is a constant;
- op is a pointwise strict operator;
- d is an identifier;
- q is an equation;
- g is a guarded definition;
- B is a expression.

We summarize each of the entries in the grammar below, and will explain each of them in detail when presenting their semantics.

- x corresponds to an *identifier.*
- c corresponds to a *constant.*
- $op(E_1, \ldots, E_n)$ is used for applying *data operators.*
- `if` E_1 `then` E_2 `else` E_3 `fi` is the *conditional operator.*
- `#`E is the *dimensional query operator*, used to probe the current context.
- $[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2}]$ is used for creating finite *context deltas.*
- $[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2},\ \texttt{all} : E_{\texttt{all}}]$ is used for creating *infinite* context deltas, all but a finite part being constant-valued.
- E_2 `@` E_1 is used for *context perturbation.*
- $(\lambda x.E)$ is used for *functional abstraction over value parameters.*
- $E_1.E_2$ is used for *application* of functions taking value parameters.
- $(\phi x.E)$ is used for *functional abstraction over named parameters.*
- $E_1\ E_2$ is used for *application* of functions taking named parameters.
- E `where` $d_{i=1..m}\ q_{j=1..n}$ is used for *local declarations* of dimensions and variables.
- $q ::= x_q = g_{i=1..n}$: An equation contains a number of guarded definitions g_i. x_q is the variable being defined.
- $g ::= B_g \to E_g$: B_g is a Boolean guard defining the context region for which this definition is valid, and E_g is the defining expression.

  The reason that the abstract syntax for the equations is quite different from the concrete syntax presented in the previous chapter is that we can imagine adding several different kinds of concrete syntax to allow the expression of context-dependent definitions of variables, however only one abstract syntax suffices.

4.2 Semantics

The first objective is to give the denotational semantics for an expression E with respect to an interpretation ζ, an environment ξ and a current context κ, written $[\![E]\!]\zeta\xi\kappa$. Before presenting this semantics, we must first introduce some notation and present the domains.

4.2.1 Functions

Suppose f is a function with finite domain given by $f(a_1) = b_1, \ldots, f(a_n) = b_n$. We will write:

$$f = \{a_i \mapsto b_i \mid i = 1..n\}.$$

Suppose f is a constant-valued function with domain S. We will write:

$$f = \{a \mapsto c \mid a \in S\}.$$

The perturbation $f \dagger g$ of a function f by a function g is given by:

$$(f \dagger g)(a) = \begin{cases} g(a), & a \in \mathrm{dom}(g) \\ f(a), & \text{otherwise.} \end{cases}$$

4.2.2 Definitions

In the semantic rules defined in the following sections, we will be manipulating values v in a domain $\mathbf{D}$, which must include three distinguished values: *true*, *false* and *init* (the latter used to initialize local dimensions). For error conditions, we will use the *undefined value* $\perp \notin \mathbf{D}$. The domain including the undefined value is $\mathbf{D}_\perp = \mathbf{D} \cup \{\perp\}$. We assume that equality can be tested on all values in $\mathbf{D}$, which we note $\equiv$. We define the flat order $\sqsubseteq$ over $\mathbf{D}_\perp$ as $v \sqsubseteq v$ and $\perp \sqsubseteq v$ for all $v \in \mathbf{D}_\perp$.

The domain $\mathbf{D}$ is the union of three sets $\mathbf{D} = \mathbf{D}_{\mathtt{const}} \cup \mathbf{D}_{\mathtt{func}} \cup \mathbf{I}$, where $\mathbf{D}_{\mathtt{const}}$ contains simple values, such as integers, strings and characters, and $\mathbf{D}_{\mathtt{func}}$, defined below, contains functions. As for $\mathbf{I}$, it is a countable set of *hidden dimensions*, which are written γ_ℓ or $\gamma_{\ell,i}$. The hidden dimensions are substituted for `where` clause local dimensions and used for implementing functions in the operational semantics.

A *context* is a total function $\kappa \in \mathbf{K}$, where $\mathbf{K} = (\mathbf{D}_{\mathtt{const}} \cup \mathbf{I}) \to \mathbf{D}_\perp$. The elements of the domain of κ are called *dimensions*. If d is a dimension, then $\kappa(d)$ is called the *d-ordinate* in κ. A context κ is called *function-free* iff $\forall d.\kappa(d) \notin \mathbf{D}_{\mathtt{func}}$.

A *context delta* is a partial function $\delta \in \Delta$, where $\Delta = (\mathbf{D}_{\mathtt{const}} \cup \mathbf{I}) \to \mathbf{D}_\perp$. Contexts will be changed by perturbing them with deltas.

The *context defined nowhere* $\kappa^\perp$ is defined by $\kappa^\perp = \{d \mapsto \perp \mid d \in (\mathbf{D}_{\mathtt{const}} \cup \mathbf{I})\}$.

An *intension* $\eta \in \mathbf{Intens}$, where $\mathbf{Intens} = \mathbf{K} \to \mathbf{D}$ is a mapping from contexts to values. If $v \in \mathbf{D}$, the constant intension ι^v is defined by $\iota^v = \{\kappa \mapsto v \mid \kappa \in \mathbf{K}\}$.

An *environment* is a function $\xi \in \mathbf{Env}$, where $\mathbf{Env} = \mathbf{Id} \to \mathbf{K} \to \mathbf{D}$. We abuse notation and extend the order $\sqsubseteq$ to environments: $\xi \sqsubseteq \xi'$ iff for all x and all κ, $\xi(x)(\kappa) \sqsubseteq \xi'(x)(\kappa)$. An environment ξ is called *function-free* iff $\forall x.\forall\kappa.\forall v$, if $\perp \neq v = \xi(x)(\kappa)$, then κ is function-free and $v \notin \mathbf{D}_{\mathtt{func}}$.

An *interpretation* is a function $\zeta \in \mathbf{Interp}$, which maps syntactic constants c to elements of $\mathbf{D}_{\mathtt{const}}$ and syntactic arity-n operators *op* to elements of $\mathbf{D}^n_{\mathtt{const}} \to \mathbf{D}_{\mathtt{const}}$.

A set of equations σ belongs to the set $\mathbf{Eqn}$. An expression E belongs to the set $\mathbf{Expr}$. The set $\mathbf{D}_{\mathtt{func}}$ is defined by:

$$\mathbf{D}_{\mathtt{func}} = \mathbf{Expr} \to \mathbf{Interp} \to \mathbf{Env} \to \mathbf{K} \to \mathbf{D}$$

4.3 Labeling abstractions and where clauses

The semantics given below assume that all functional parameters and local variables are unique, and that local dimension identifiers are replaced by hidden dimensions. To ensure this, a *labeling* process, using substitution, is defined, to produce a slightly different abstract syntax.

4.3.1 Identifiers and substitutions

Let $\mathbf{X}$ be the set of identifiers that may appear in a program. We will write X for a subset of $\mathbf{X}$. In addition to these, we add a countable set of *hidden identifiers* $\mathbf{Y}$, which are written $\chi_0, \chi_1, \ldots$. The hidden identifiers are substituted for local variables when `where` clauses are encountered upon evaluation, or for input parameters in function applications.

The semantics requires the ability to substitute a set of identifiers by expressions. A substitution φ of a finite set of identifiers x_1 to E_1, …, x_n to E_n is written $[x_i/E_i \mid i = 1..n]$. In the semantic rules below, all substitutions are for constant values or for hidden variables.

To define the application of a substitution φ to an expression E or an equation q, written $E\varphi$ or $q\varphi$, respectively, we must first define some notation:

$$\varphi\backslash X = \big[x/E \mid x/E \in \varphi \wedge x \notin X\big]$$

Here is the rule for substitution in an equation:

$$(x_q = g_{i=1..n})\varphi = x_q\varphi = g_{i=1..n}\varphi$$

Here is the rule for substitution in a guard:

$$(B_g \to E_g)\varphi = B_g\varphi \to E_g\varphi$$

Here are the rules for substitution in an expression:

$$\begin{aligned}
x\varphi &= \begin{cases} E & x/E \in \varphi \\ x & \text{otherwise} \end{cases} \\
c\varphi &= c \\
\big(op(E_1, \ldots, E_n)\big)\varphi &= op(E_1\varphi, \ldots, E_n\varphi) \\
(\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ fi})\varphi &= \texttt{if } E_1\varphi \texttt{ then } E_2\varphi \texttt{ else } E_3\varphi \texttt{ fi} \\
(\texttt{\#}E)\varphi &= \texttt{\#}(E\varphi) \\
\big([E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2}]\big)\varphi &= [E_{11}\varphi : E_{12}\varphi,\ \ldots,\ E_{n1}\varphi : E_{n2}\varphi] \\
\big([E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2},\ \texttt{all} : E_{\texttt{all}}]\big)\varphi &= [E_{11}\varphi : E_{12}\varphi,\ \ldots,\ E_{n1}\varphi : E_{n2}\varphi,\ \texttt{all} : E_{\texttt{all}}\varphi] \\
(E_2 \ \texttt{@}\ E_1)\varphi &= (E_2\varphi)\ \texttt{@}\ (E_1\varphi) \\
(\lambda x.E)\varphi &= \text{let } \ \varphi' = \varphi\backslash\{x\} \ \text{ in } (\lambda x.E\varphi') \\
(E_1.E_2)\varphi &= (E_1\varphi).(E_2\varphi) \\
(\phi x.E)\varphi &= \text{let } \ \varphi' = \varphi\backslash\{x\} \ \text{ in } (\phi x.E\varphi') \\
(E_1\ E_2)\varphi &= (E_1\varphi)\ (E_2\varphi) \\
(E \texttt{ where } d_{i=1..m}\ q_{j=1..n})\varphi &= \text{let } \ \varphi' = \varphi\backslash\big(\{d_i \mid i = 1..n\} \cup \{x_{q_j} \mid i = 1..n\}\big) \ \text{ in} \\
&\quad\ E\varphi' \texttt{ where } d_{i=1..n}\varphi'\ q_{j=1..n}\varphi'
\end{aligned}$$

4.3.2 Labeling

A unique label ℓ is given to each λ-abstraction, ϕ-abstraction and `where` clause. The choice of labeling is not important. Then, statically, from the top down, each of these is replaced by an alternative expression, given below. The semantics given below are for the rewritten expressions.

- $(\lambda x.E)$ is replaced by $(\lambda_\ell\ E\varphi)$, where $\varphi = [x/\chi_\ell]$.
- $(\phi x.E)$ is replaced by $(\phi_\ell\ E\varphi)$, where $\varphi = [x/\chi_\ell]$.
- E `where` $d_{i=1..m}\ q_{j=1..n}$ is replaced by $E\varphi$ $\texttt{where}_\ell$ $\gamma_{\ell,i=1..m}\ \ q_{j=1..n}\varphi$, where

$$\varphi = \left[d_i/\gamma_{\ell,i},\ x_{q_k}/\chi_{\ell,k} \mid i = 1..m, k = 1..n\right]$$

  Since local definitions allow the use of local, named dimensions. we need to have new dimensions for each of these names. The hidden dimensions and variables are all numbered.

4.4 Denotational semantics

4.4.1 Semantics of expressions

We need to give the semantics for an expression E with respect to an interpretation ζ, an environment ξ and a current running context κ in order to produce a value v. We write this

$$v = [\![E]\!]\zeta\xi\kappa$$

i.e., $[\![\cdot]\!] \in \mathbf{D}_{\texttt{func}}$. The semantic rules for an expression E are built up through the structure of E. These rules are parameterized by the predicates *moreSpecific* and *accessible*, by the constant *init* and by the operator $\oplus$; each of these is explained as it is encountered below.

The presentation of the semantics will require the labeling of all of the functions and local definitions. Therefore, we will be giving semantics to $(\lambda_\ell\, E)$ instead of $(\lambda x.E)$, to $(\phi_\ell\, E)$ instead of $(\phi x.E)$, and to E $\texttt{where}_\ell$ $\gamma_{\ell,i=1..m}\ q_{j=1..n}$ instead of E `where` $d_{i=1..m}\ q_{j=1..n}$.

Identifiers

x: When an identifier x is encountered, we look it up in the environment ξ and then apply the result to κ:

$$[\![x]\!]\zeta\xi\kappa = \xi(x)(\kappa)$$

Constants

c: A constant c is constant in all contexts.

$$[\![c]\!]\zeta\xi\kappa = \zeta(c)$$

Operations

$op(E_1, \ldots, E_n)$: When an operation is to be evaluated, the operands are all evaluated in the same context; then the operator is applied to these, also in that context.

$$\left[\!\left[op(E_1, \ldots, E_n)\right]\!\right]\zeta\xi\kappa = \zeta(op)\big([\![E_1]\!]\zeta\xi\kappa, \ldots, [\![E_n]\!]\zeta\xi\kappa\big)$$

Conditionals

`if` E_1 `then` E_2 `else` E_3 `fi`: When a conditional expression is to be evaluated, the condition E_1 is evaluated in the same context κ; then, depending on the returned value v_1, one of the choices E_2 or E_3 is to be evaluated, also in κ.

$$\begin{aligned} & [\![\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ fi}]\!]\zeta\xi\kappa \\ = {} & \text{let } \ v_1 = [\![E_1]\!]\zeta\xi\kappa \ \text{ in} \\ & \begin{cases} [\![E_2]\!]\zeta\xi\kappa, & v_1 \equiv \mathit{true} \\ [\![E_3]\!]\zeta\xi\kappa, & v_1 \equiv \mathit{false} \end{cases} \end{aligned}$$

Context query

`#`E: The context query requires evaluating the expression E to get a value v. The context κ is then applied to v, giving the v-ordinate of κ.

$$[\![\texttt{\#}E]\!]\zeta\xi\kappa = \kappa\big([\![E]\!]\zeta\xi\kappa\big)$$

Finite deltas

$[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2}]$: The finite delta requires evaluating the $2n$ subexpressions E_{ij}, then returning a finite function built up from the results of these.

$$\begin{aligned}&[\![[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2}]]\!]\zeta\xi\kappa\\ =\ &\text{let}\ \ v_{ij} = [\![E_{ij}]\!]\zeta\xi\kappa,\ i = 1..n, j = 1..2\ \ \text{in}\\ &\{v_{i1} \mapsto v_{i2} \mid i = 1..n\}\end{aligned}$$

Infinite deltas

$[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2},\ \texttt{all} : E_{\texttt{all}}]$: The infinite delta contains two components, a constant context and a finite delta used to perturb that constant context. Should an infinite delta be used to perturb a context, then that delta will completely replace the current context.

$$\begin{aligned}&[\![[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2},\ \texttt{all} : E_{\texttt{all}}]]\!]\zeta\xi\kappa\\ =\ &\text{let}\ \ v_{ij} = [\![E_{ij}]\!]\zeta\xi\kappa,\ i = 1..n, j = 1..2\ \ \text{in}\\ =\ &\text{let}\ \ v_{\texttt{all}} = [\![E_{\texttt{all}}]\!]\zeta\xi\kappa\ \ \text{in}\\ &\big\{v \mapsto v_{\texttt{all}} \mid v \in (\mathbf{D}_{\texttt{const}} \cup \mathbf{I})\big\} \dagger \{v_{i1} \mapsto v_{i2} \mid i = 1..n\}\end{aligned}$$

Context perturbation

E_2 `@` E_1: Context perturbation requires evaluating E_1 to produce a delta, which is then used as new running context for the evaluation of E_2.

$$\begin{aligned}&[\![E_2\ \texttt{@}\ E_1]\!]\zeta\xi\kappa\\ =\ &\text{let}\ \ \delta = [\![E_1]\!]\zeta\xi\kappa\ \ \text{in}\\ &[\![E_2]\!]\zeta\xi(\kappa \dagger \delta), \qquad \mathit{accessible}(\kappa, \kappa \dagger \delta)\end{aligned}$$

The default *accessible* function yields *true* in all circumstances. Once time is introduced, *accessible* prevents accessing the future. In physically constrained systems, *accessible* could be defined to take into account other dimensions.

Value-parameter application

The value-parameter functions are evaluated using call-by-value. The actual parameter is fully evaluated to produce a value v, from which is created the constant intension ι^v. The body of the function is evaluated with the meaning of χ_ℓ defined to be ι^v. Note that the context κ is bound dynamically while the environment ξ is bound statically. The meaning of the function is a member of $\mathbf{D}_{\texttt{func}}$.

$$\begin{aligned}[\![(\lambda_\ell\, E)]\!]\zeta\xi\kappa = \ &\lambda E'.\lambda\zeta'.\lambda\xi'.\lambda\kappa'.\\ &\text{let}\ \ v = [\![E']\!]\zeta'\xi'\kappa'\ \ \text{in}\\ &[\![E]\!]\zeta\big(\xi[\chi_\ell/\iota^v]\big)\kappa'\end{aligned}$$

The application of a function simply consists of applying the meaning of the function to the argument.

$$[\![E_1.E_2]\!]\zeta\xi\kappa = [\![\big([\![E_1]\!]\zeta\xi\kappa\big)(E_2)]\!]\zeta\xi\kappa$$

Named-parameter application

The named-parameter functions are evaluated using call-by-name. Since the entities being passed as actual parameters are typically infinite, multidimensional entities, they could never be fully evaluated anyway. The meaning of χ_ℓ is defined to be the intension η, which is defined to be $[\![E]\!]\zeta'\xi'$, not $[\![E]\!]\zeta'\xi'\kappa'$. The meaning of the function is a member of $\mathbf{D}_{\mathtt{func}}$.

$$\begin{aligned}[\![(\phi_\ell\ E)]\!]\zeta\xi\kappa = {} & \lambda E'.\lambda\zeta'.\lambda\xi'.\lambda\kappa'. \\ & \text{let } \ \eta = [\![E']\!]\zeta'\xi' \ \text{ in} \\ & [\![E]\!]\zeta\big(\xi[\chi_\ell/\eta]\big)\kappa'\end{aligned}$$

The application of a function simply consists of applying the meaning of the function to the argument.

$$[\![E_1\ E_2]\!]\zeta\xi\kappa = \big[\!\big[\big([\![E_1]\!]\zeta\xi\kappa\big)(E_2)\big]\!\big]\zeta\xi\kappa$$

Local definitions

The meaning of the `where` clause is given using fixed points. A sequence of environments ξ_i, $i \in \mathbb{N}$, is defined by initializing $\xi_0 = \xi$, then applying the meaning of the individual equations to produce ξ_{i+1} from ξ_i. The expression E is then evaluated in the least fixpoint environment ξ' resulting from the sequence of the ξ_i.

A new dimension must be set to the *init* value as soon as it is created.

$$\begin{aligned} & [\![E \ \mathtt{where}_\ell \ \gamma_{\ell,i=1..m} \ q_{j=1..n}]\!]\zeta\xi\kappa \\ = {} & \text{let } \ \kappa' = \kappa \dagger \{\gamma_{\ell,i} \mapsto \mathit{init} \mid i = 1..m\} \ \text{ in} \\ & \text{let } \ \xi_0 = \xi \ \text{ in} \\ & \text{let } \ \xi_{i+1} = \xi_i\big[\chi_{\ell,j}/[\![q_j]\!]\zeta\xi_i \mid j = 1..n\big] \ \text{ in} \\ & \text{let } \ \xi' = \operatorname{lfp} \xi_i \ \text{ in} \\ & [\![E]\!]\zeta\xi'\kappa' \end{aligned}$$

The fixpoint exists because the $[\![E]\!]$ is continuous for all E, hence monotonic.

Equations and bestfitting

An equation q will define a variable x_q through a set of guarded definitions. We must look for the bestfit definitions with respect to the current context κ, evaluate them all, then combine them with some associative, commutative operator $\oplus$.

$$\begin{aligned}[\![q]\!]\zeta\xi\kappa = \text{let } \{g_1,\ldots,g_n\} = {} & \big\{g \in q \mid [\![B_g]\!]\zeta\xi\kappa \equiv \mathit{true} \wedge \\ & \quad (\nexists g' \in q)\ \mathit{moreSpecific}(B_{g'}, B_g)\big\} \text{ in} \\ \text{let } \ v_i = {} & [\![E_{g_i}]\!]\zeta\xi\kappa \ \text{ in} \\ v_1 \oplus \cdots \oplus v_n & \end{aligned}$$

where $\mathit{moreSpecific}(B_{g'}, B_g)$ means that the region $\{\kappa \mid [\![B_{g'}]\!]\zeta\xi\kappa \equiv \mathit{true}\}$ is *provably* (computationally) *strictly smaller* than the region $\{\kappa \mid [\![B_g]\!]\zeta\xi\kappa \equiv \mathit{true}\}$.

Typical examples of $\oplus$ are uniqueness ($\bot$ if $n > 1$), determinacy ($\bot$ if $(\exists i \neq j)\ v_i \neq v_j$), median, mean, mode, average, sum, product, and set union. In the implementation, $\oplus$ can be specified separately for each variable.

We consider both the predicate *moreSpecific* and the $\oplus$ operator to be parameters of the semantic rules.

4.5 Operational semantics

4.5.1 Semantics of expressions

We need to give the semantics for an expression E with respect to an interpretation ζ, an environment ξ, set of equations σ and a current running context κ in order to produce a value v. We write this

$$v = \langle\!\langle E \rangle\!\rangle \zeta\xi\sigma\kappa$$

However, there are subtleties that are not shown by the syntax. In particular, the meaning of $\mathbf{D}_{\texttt{func}}$ (not $\mathbf{D}_{\texttt{const}}$) is slighly different. Instead of

$$\mathbf{D}_{\texttt{func}} = \mathbf{Expr} \to \mathbf{Interp} \to \mathbf{Env} \to \mathbf{K} \to \mathbf{D},$$

the meaning is

$$\mathbf{D}^{\langle\!\langle \cdot \rangle\!\rangle}_{\texttt{func}} = \mathbf{Expr} \to \mathbf{Interp} \to \mathbf{Env} \to \mathbf{Eqn} \to \mathbf{K} \to \mathbf{D},$$

and all of the domains should then be superscripted by $\langle\!\langle \cdot \rangle\!\rangle$.

$$\begin{aligned}
\mathbf{D}^{\langle\!\langle \cdot \rangle\!\rangle} &= \mathbf{D}_{\texttt{const}} \cup \mathbf{D}^{\langle\!\langle \cdot \rangle\!\rangle}_{\texttt{func}} \\
\mathbf{K}^{\langle\!\langle \cdot \rangle\!\rangle} &= (\mathbf{D}_{\texttt{const}} \cup \mathbf{I}) \to \mathbf{D}^{\langle\!\langle \cdot \rangle\!\rangle}_{\perp} \\
\Delta^{\langle\!\langle \cdot \rangle\!\rangle} &= (\mathbf{D}_{\texttt{const}} \cup \mathbf{I}) \to \mathbf{D}^{\langle\!\langle \cdot \rangle\!\rangle}_{\perp} \\
\mathbf{Intens}^{\langle\!\langle \cdot \rangle\!\rangle} &= \mathbf{K}^{\langle\!\langle \cdot \rangle\!\rangle} \to \mathbf{D}^{\langle\!\langle \cdot \rangle\!\rangle} \\
\mathbf{Env}^{\langle\!\langle \cdot \rangle\!\rangle} &= \mathbf{Id} \to \mathbf{K}^{\langle\!\langle \cdot \rangle\!\rangle} \to \mathbf{D}^{\langle\!\langle \cdot \rangle\!\rangle}
\end{aligned}$$

We therefore have $\langle\!\langle \cdot \rangle\!\rangle \in \mathbf{D}^{\langle\!\langle \cdot \rangle\!\rangle}_{\texttt{func}}$. We will avoid this superscripting when it is not necessary.

The environment ξ is not updated by the rules; it is used to provide meaning to free variables in E. The rules for an expression E are built up through the structure of E. These rules are parameterized, like for the denotational semantics, by the predicates *moreSpecific* and *accessible*, by the constant *init* and by the operator $\oplus$; each of these is explained as it is encountered below.

Identifiers

x: When an identifier x is encountered, we lookup its equation in σ, then evaluate that equation:

$$\langle\!\langle x \rangle\!\rangle \zeta\xi\sigma\kappa = \begin{cases} \xi(x)(\kappa), & \text{if } x \in \mathrm{dom}(\xi) \\ \langle\!\langle \sigma(x) \rangle\!\rangle \zeta\xi\sigma\kappa, & \text{otherwise} \end{cases}$$

Constants

c: A constant c is constant in all contexts.

$$\langle\!\langle c \rangle\!\rangle \zeta\xi\sigma\kappa = \zeta(c)$$

Operations

$op(E_1, \ldots, E_n)$: When an operation is to be evaluated, the operands are all evaluated in the same context; then the operator is applied to these, also in that context.

$$\langle\!\langle op(E_1, \ldots, E_n) \rangle\!\rangle \zeta\xi\sigma\kappa = \zeta(op)\big(\langle\!\langle E_1 \rangle\!\rangle \zeta\xi\sigma\kappa, \ldots, \langle\!\langle E_n \rangle\!\rangle \zeta\xi\sigma\kappa\big)$$

Conditionals

`if` E_1 `then` E_2 `else` E_3 `fi`: When a conditional expression is to be evaluated, the condition E_1 is evaluated in the same context κ; then, depending on the returned value v_1, one of the choices E_2 or E_3 is to be evaluated, also in κ.

$$\begin{aligned}
&\langle\!\langle \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\ \texttt{fi} \rangle\!\rangle \zeta\xi\sigma\kappa \\
&= \text{let}\ \ v_1 = \langle\!\langle E_1 \rangle\!\rangle \zeta\xi\sigma\kappa\ \ \text{in} \\
&\quad \begin{cases} \langle\!\langle E_2 \rangle\!\rangle \zeta\xi\sigma\kappa, & v_1 \equiv \textit{true} \\ \langle\!\langle E_3 \rangle\!\rangle \zeta\xi\sigma\kappa, & v_1 \equiv \textit{false} \end{cases}
\end{aligned}$$

Context query

`#`E: The context query requires evaluating the expression E to get a value v. The context κ is then applied to v, giving the v-ordinate of κ.

$$\langle\!\langle \texttt{\#}E \rangle\!\rangle \zeta\xi\sigma\kappa = \kappa\big(\langle\!\langle E \rangle\!\rangle \zeta\xi\sigma\kappa\big)$$

Finite deltas

$[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2}]$: The finite delta requires evaluating the $2n$ subexpressions E_{ij}, then returning a finite function built up from the results of these.

$$\begin{aligned}
&\langle\!\langle [E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2}] \rangle\!\rangle \zeta\xi\sigma\kappa \\
&= \text{let}\ \ v_{ij} = \langle\!\langle E_{ij} \rangle\!\rangle \zeta\xi\sigma\kappa,\ i = 1..n, j = 1..2\ \ \text{in} \\
&\quad \{v_{i1} \mapsto v_{i2} \mid i = 1..n\}
\end{aligned}$$

Infinite deltas

$[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2},\ \texttt{all} : E_{\texttt{all}}]$: The infinite delta contains two components, a constant context and a finite delta used to perturb that constant context. Should an infinite delta be used to perturb a context, then that delta will completely replace the current context.

$$\begin{aligned}
&\langle\!\langle [E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2},\ \texttt{all} : E_{\texttt{all}}] \rangle\!\rangle \zeta\xi\sigma\kappa \\
&= \text{let}\ \ v_{ij} = \langle\!\langle E_{ij} \rangle\!\rangle \zeta\xi\sigma\kappa,\ i = 1..n, j = 1..2\ \ \text{in} \\
&= \text{let}\ \ v_{\texttt{all}} = \langle\!\langle E_{\texttt{all}} \rangle\!\rangle \zeta\xi\sigma\kappa\ \ \text{in} \\
&\quad \{v \mapsto v_{\texttt{all}} \mid v \in (\mathbf{D}_{\texttt{const}} \cup \mathbf{I})\} \dagger \{v_{i1} \mapsto v_{i2} \mid i = 1..n\}
\end{aligned}$$

Context perturbation

E_2 `@` E_1: Context perturbation requires evaluating E_1 to produce a delta, which is then used as new running context for the evaluation of E_2.

$$\begin{aligned}
&\langle\!\langle E_2\ \texttt{@}\ E_1 \rangle\!\rangle \zeta\xi\sigma\kappa \\
&= \text{let}\ \ \delta = \langle\!\langle E_1 \rangle\!\rangle \zeta\xi\sigma\kappa\ \ \text{in} \\
&\quad \langle\!\langle E_2 \rangle\!\rangle \zeta\xi\sigma(\kappa \dagger \delta), \qquad \textit{accessible}(\kappa, \kappa \dagger \delta)
\end{aligned}$$

Value-parameter application

The definitions for function applications in the denotational semantics given in §4.4 explicitly manipulate the environment ξ, which makes them difficult to translate into operational semantics. By introducing, for each function, a new hidden dimension, we can transform function applications into context changes. This work is inspired by work by Panos Rondogiannis and William

Wadge [81, 82]. What is different here is that each function will get its own dimension instead of using one dimension for all functions or for all functions of the same order.

The denotational semantics gives:

$$\begin{aligned}
[\![(\lambda_\ell\, E)]\!]\zeta\xi\kappa = {} & \lambda E'.\lambda\zeta'.\lambda\xi'.\lambda\kappa'. \\
& \text{let } \ v = [\![E']\!]\zeta'\xi'\kappa' \ \text{ in} \\
& [\![E]\!]\zeta\big(\xi[\chi_\ell/\iota^v]\big)\kappa'
\end{aligned}$$

$$[\![E_1.E_2]\!]\zeta\xi\kappa = \big[\!\big[\big([\![E_1]\!]\zeta\xi\kappa\big)(E_2)\big]\!\big]\zeta\xi\kappa$$

We handle a value-parameter application as follows:

- We use γ_ℓ as a new dimension, for all occurrences of this, and only this, function.
- We add a single new equation $\chi_\ell \mid \mathit{true} \rightarrow \#\gamma_\ell$ to the environment σ in which the abstraction is evaluated.
- We add a single entry $\{\gamma_\ell \mapsto \langle\!\langle E'\rangle\!\rangle\zeta'\xi'\sigma'\kappa'\}$ to the context κ' in which the argument is evaluated.
- We then evaluate the body E.

Here are the rules for value-parameter application:

$$\begin{aligned}
\langle\!\langle(\lambda_\ell\, E)\rangle\!\rangle\zeta\xi\sigma\kappa = {} & \lambda E'.\lambda\zeta'.\lambda\xi'.\lambda\sigma'.\lambda\kappa'. \\
& \text{let } \ \sigma'' = \sigma \cup \{\chi_\ell \mid \mathit{true} \rightarrow \#\gamma_\ell\} \ \text{ in} \\
& \text{let } \ \kappa'' = \kappa' \dagger \{\gamma_\ell \mapsto \langle\!\langle E'\rangle\!\rangle\zeta'\xi'\sigma'\kappa'\} \ \text{ in} \\
& \langle\!\langle E\rangle\!\rangle\zeta\xi\sigma''\kappa''
\end{aligned}$$

$$\langle\!\langle E_1.E_2\rangle\!\rangle\zeta\xi\sigma\kappa = \big\langle\!\big\langle\big(\langle\!\langle E_1\rangle\!\rangle\zeta\xi\sigma\kappa\big)(E_2)\big\rangle\!\big\rangle\zeta\xi\sigma\kappa$$

To see that this transformation is valid, consider the evaluation of χ_ℓ inside E, with a current context κ_{in}. Then

$$\kappa_{\text{in}}(\gamma_\ell) = \langle\!\langle E'\rangle\!\rangle\zeta'\xi'\sigma'\kappa'$$

since dimension γ_ℓ can only be changed, by definition, by an invocation of λ_ℓ. Then the value of χ_ℓ inside the evaluation of E will yield:

$$\langle\!\langle\chi_\ell\rangle\!\rangle\zeta\xi\sigma''\kappa_{\text{in}} = \langle\!\langle\#\gamma_\ell\rangle\!\rangle\zeta\xi\sigma''\kappa_{\text{in}} = v$$

which is exactly the value that χ_ℓ would have in $\xi[\chi_\ell/\iota^v]$. Hence the transformation is valid.

Named-parameter application

The denotational semantics gives:

$$\begin{aligned}
[\![(\phi_\ell\, E)]\!]\zeta\xi\kappa = {} & \lambda E'.\lambda\zeta'.\lambda\xi'.\lambda\kappa'. \\
& \text{let } \ \eta = [\![E']\!]\zeta'\xi' \ \text{ in} \\
& [\![E]\!]\zeta\big(\xi[\chi_\ell/\eta]\big)\kappa'
\end{aligned}$$

$$[\![E_1\ E_2]\!]\zeta\xi\kappa = \big[\!\big[\big([\![E_1]\!]\zeta\xi\kappa\big)(E_2)\big]\!\big]\zeta\xi\kappa$$

We handle a named-parameter application as follows:

- We use γ_ℓ as a new dimension, for all occurrences of this, and only this, function.
- All expressions, upto renaming of variables, are tagged with unique labels. In this case, we tag E' with j.
- We add equation $\chi_\ell \mid hd(\#\gamma_\ell) \equiv j \rightarrow E' \mathbin{@} [\gamma_\ell : tl(\#\gamma_\ell)]$ to the environment σ in which the abstraction is evaluated. There is a new equation for χ_ℓ for each invocation of the function.
- We add a single entry $\{\gamma_\ell \mapsto cons(j, \kappa'(\gamma_\ell))\}$ to the context κ' in which the argument is evaluated.
- We then evaluate the body E.

Here are the rules for named-parameter application:

$$\begin{aligned}
\langle\!\langle (\phi_\ell\, E) \rangle\!\rangle \zeta\xi\sigma\kappa = {} & \lambda E'.\lambda\zeta'.\lambda\xi'.\lambda\sigma'.\lambda\kappa'. \\
& \text{let } \sigma'' = \sigma \cup \left\{\chi_\ell \mid hd(\#\gamma_\ell) \equiv j \rightarrow E' \mathbin{@} [\gamma_\ell : tl(\#\gamma_\ell)]\right\} \text{ in} \\
& \text{let } \kappa'' = \kappa' \dagger \left\{\gamma_\ell \mapsto cons(j, \kappa'(\gamma_\ell))\right\} \text{ in} \\
& \langle\!\langle E \rangle\!\rangle \zeta\xi\sigma''\kappa''
\end{aligned}$$

$$\langle\!\langle E_1\ E_2 \rangle\!\rangle \zeta\xi\sigma\kappa = \langle\!\langle \big(\langle\!\langle E_1 \rangle\!\rangle \zeta\xi\sigma\kappa\big)(E_2) \rangle\!\rangle \zeta\xi\sigma\kappa$$

To see that this transformation is valid, consider the evaluation of χ_ℓ inside E, with a current context κ_{in}. Then

$$\kappa_{\text{in}}(\gamma_\ell) = cons(j, \kappa(\gamma_\ell))$$

since dimension γ_ℓ can only be changed, by definition, by an invocation of λ_ℓ. Since $\text{hd}(\kappa_{\text{in}}) \equiv j$, the value of χ_ℓ inside the evaluation of E will yield:

$$\begin{aligned}
\langle\!\langle \chi_\ell \rangle\!\rangle \zeta\xi\sigma''\kappa_{\text{in}} &= \langle\!\langle E' \mathbin{@} [\gamma_\ell : tl(\#\gamma_\ell)] \rangle\!\rangle \zeta\xi\sigma''\kappa_{\text{in}} \\
&= \langle\!\langle E' \rangle\!\rangle \zeta\xi\sigma'' \big(\kappa_{\text{in}} \dagger \{\gamma_\ell \mapsto tl(cons(j, \kappa(\gamma_\ell)))\}\big) \\
&= \langle\!\langle E' \rangle\!\rangle \zeta\xi\sigma'' \big(\kappa_{\text{in}} \dagger \{\gamma_\ell \mapsto \kappa(\gamma_\ell)\}\big)
\end{aligned}$$

which corresponds exacly to the value that χ_ℓ would have in $\xi[\chi_\ell/\eta]$.

Local definitions

The local definitions are handled by augmenting the set of equations, renaming the variables to avoid any name clashes.

$$\begin{aligned}
& \langle\!\langle E \ \texttt{where}_\ell\ \gamma_{\ell,i=1..m}\ q_{j=1..n} \rangle\!\rangle \zeta\xi\sigma\kappa \\
= {} & \text{let } \kappa' = \kappa \dagger \{\gamma_{\ell,i} \mapsto init \mid i = 1..m\} \text{ in} \\
& \text{let } \sigma' = \sigma \cup \{q_j \mid j = 1..n] \text{ in} \\
& \langle\!\langle E \rangle\!\rangle \zeta\xi\sigma'\kappa'
\end{aligned}$$

Equations and bestfitting

$$\begin{aligned}
\langle\!\langle q \rangle\!\rangle \zeta\xi\sigma\kappa = {} & \text{let } \{g_1, \ldots, g_n\} = \ \big\{g \in q \mid \langle\!\langle B_g \rangle\!\rangle \zeta\xi\sigma\kappa \equiv true \wedge \\
& \qquad\qquad\qquad\qquad (\nexists g' \in q)\ moreSpecific(B_{g'}, B_g)\big\} \text{ in} \\
& \text{let } v_i = \langle\!\langle E_{g_i} \rangle\!\rangle \zeta\xi\sigma\kappa \text{ in} \\
& v_1 \oplus \cdots \oplus v_n
\end{aligned}$$

where $moreSpecific(B_{g'}, B_g)$ means that the region $\{\kappa \mid \langle\!\langle B_{g'} \rangle\!\rangle \zeta\xi\sigma\kappa \equiv true\}$ is *provably* (computationally) *strictly smaller* than the region $\{\kappa \mid \langle\!\langle B_g \rangle\!\rangle \zeta\xi\sigma\kappa \equiv true\}$.

4.5.2 Results

Proposition 10. *Let ξ be a function-free environment and κ be a function-free context. Then $[\![E]\!]\zeta\xi\kappa = v$ iff $\langle\!\langle E\rangle\!\rangle\zeta\xi\varnothing\kappa = v$, where $v \in \mathbf{D}_{\texttt{const}}$.*

The restriction on ξ is to ensure that it can be considered to be an element both of **Env** and $\mathbf{Env}^{\langle\!\langle\cdot\rangle\!\rangle}$. Similarly for κ, **K** and $\mathbf{K}^{\langle\!\langle\cdot\rangle\!\rangle}$. The proof is a straightforward double-induction, first on the depth of `where` clause-nesting, second on the number of iterations within a `where` clause. The denotational semantics computes bottom-up while the operational semantics computes top-down. Nevertheless, if the denotational semantics yields a value $v \neq \bot$, there will be a natural number i for which i iterations are sufficient to compute v. The depth of the tree of the operational semantics will also be i. The reverse argument holds as well.

4.6 Caching

The operational semantics presented in the previous section would lead naturally to an inefficient solution. To improve the result, we can use caching of the intermediate results, i.e., store (x, κ, v) triples in a cache to avoid recomputation. However, given that κ is very large, potentially infinite, and that most of the dimensions in κ will never even be examined, it is necessary to take an approach in which the caching is with respect to the portion of κ that has actually been *examined* during computation.

Doing this, however, makes it more difficult to probe the cache. The solution is to begin the evaluation of an identifier with an empty context, and to evaluate the expression; should the result be that dimensions are missing, this information is stored in the cache, these are added to the context, and the expression is reevaluated. This process is repeated until the expression can be evaluated completely and the returned value can be cached.

This "game" approach to caching was first discussed by Anthony Faustini and William Wadge when they were writing the pLucid interpreter in the mid-1980s [38]; however they did not implement this caching mechanism. A sequential implementation of this caching mechanism was developed by myself in 2007, after discussion with William Wadge [75].

The semantics presented here is to support a multi-threaded implementation, with the intuition derived from the TransLucid implementation written by Toby Rahilly in 2009 [80]. For the evaluation of tuples and of operations, multiple threads can be spawned, and each of these may interact with the cache. To properly interleave the actions of each thread upon the cache, a notion of time must be introduced into the behavior of the threads.

A number of different kinds of value are manipulated by this semantics, so it is easier to type the values being manipulated. This also gives us an opportunity to distinguish the different kinds of error situation.

4.6.1 Typed values

A value v is defined to be a pair $(\tau(v), \psi(v))$, where $\tau(v)$ is the *type* of v and $\psi(v)$ is the *concrete value* of v. When writing down explicitly v, we can write $\tau(v)\langle\psi(v)\rangle$.

The semantics presented here explicitly manipulates the following types:

- `bool`: Booleans. The concrete values are `true` and `false`.
- `sp`: Special values to deal with exceptional or erroneous situations. The concrete values are:
  - `undef`: undefined variable;
  - `type`: type error; and
  - `tuple`: inconsistent tuple construction;
  - `loop`: infinite loop detected.

To ensure that specials are used in a deterministic manner, they are ordered, with `undef` $\leqslant$ `type` $\leqslant$ `tuple` $\leqslant$ `loop`. The operation *minsp* selects the minimum concrete special value from a set.

- `tuple`: Tuple values used to hold deltas. The concrete values are functions from dimensions to ordinates.
- `calc`: Worker threads. The concrete values are lists of natural numbers, written w. A thread w may generate several worker threads $w_i = \mathit{cons}(i, w)$, $i = 1..n$. Should this be the case, we write $w_i \geqslant w$.
- `dem`: Demands. The concrete values are of the form $\texttt{dem}\langle v_1, \ldots, v_n\rangle$, which corresponds to a demand for the dimensions v_1 through to v_n.

4.6.2 The cache

The field β continues to provide the environment, this time in the form of a cache. There are two data structures and two operations:

- $\beta.\mathit{time} \in \mathbb{N}$ is a timestamp. Timestamps can be incremented (+1), they can be compared ($\leqslant$) and their maximum *max* can be computed.
- $\beta.\mathit{cache} \in \mathbf{X} \times \mathbf{K} \times \mathbb{N} \to \mathbf{V}$, which maps (*identifier*, *delta*, *timestamp*) triples to atomic values, such that none of the timestamps may be greater than the current $\beta.\mathit{time}$.
- $\beta.\text{find}(x, \delta, t, w)$, with $t \leqslant \beta.\mathit{time}$, yields (v, t'), with $t' \geqslant t$, and is defined as follows:

```
Case $\exists t' \geqslant t.(x, \delta, t') \in \text{dom}(\beta.\mathit{cache})$:
    let $t_v = \max\{t_i \mid (x, \delta, t_i) \in \text{dom}(\beta.\mathit{cache})\}$;
    let $v = \beta.\mathit{cache}(x, \delta, t_v)$;
    return $\big(v, \max(t, t_v)\big)$.
Case $\nexists t' \geqslant t.(x, \delta, t') \in \text{dom}(\beta.\mathit{cache})$:
    $\beta.\mathit{time} := \beta.\mathit{time} + 1$;
    $\beta.\mathit{cache} := \beta.\mathit{cache} \cup \big\{(x, \delta, \beta.\mathit{time}) \mapsto \texttt{calc}\langle w\rangle\big\}$;
    return $\big(\texttt{calc}\langle w\rangle, \beta.\mathit{time}\big)$.
```

  In the second case, should several threads $w_1, \ldots, w_n$ all attempt to access the same location simultaneously, then one thread w' would be nondeterministically chosen to be calculating, and all of these threads, including w', would receive the value $\texttt{calc}\langle w'\rangle$.

- $\beta.\text{add}(x, \kappa, t, v)$, with $t \leqslant \beta.\text{time}$, yields t', with $t' \geqslant t$, and defined as follows:

```
    $\beta.\mathit{time} := \beta.\mathit{time} + 1$;
    $\beta.\mathit{cache} := \beta.\mathit{cache} \cup \big\{(x, \kappa, \beta.\mathit{time}) \mapsto v\big\}$;
    return $\beta.\mathit{time}$.
```

The contents of the cache β are designed to work as follows, should all of the calculations for a request have been made.

$$\begin{array}{rcl}
\beta.\text{find}(x, \varnothing, t, w) & = & \big(\texttt{dem}\langle V_1\rangle, t\big) \\
\beta.\text{find}(x, \kappa \mid V_1, t, w) & = & \big(\texttt{dem}\langle V_2\rangle, t\big) \\
\beta.\text{find}\big(x, \kappa \mid (V_1 \cup V_2), t, w\big) & = & \big(\texttt{dem}\langle V_3\rangle, t\big) \\
 & \cdots & \\
\beta.\text{find}\big(x, \kappa \mid (V_1 \cup \cdots \cup V_{n-1}), t, w\big) & = & \big(\texttt{dem}\langle V_n\rangle, t\big) \\
\beta.\text{find}\big(x, \kappa \mid (V_1 \cup \cdots \cup V_n), t, w\big) & = & (v, t)
\end{array}$$

where $\texttt{dem}\langle V_i\rangle$ means a demand for the dimensions in V_i. We will write $\beta^*(x, \kappa, t, w) = v$ iff there is a chain of V_i as above. Furthermore, the initial environment has to be set up so that the inputs are inserted in this manner.

4.6.3 Rules

These rules do not work with full contexts κ, but with deltas δ. The rules are of the form $(v, t') = [\![E]\!]^*\zeta\beta\sigma\delta w t$, which means that expression E is evaluated using interpretation ζ and cache β in a context delta δ by a worker thread w at time t, yielding value v at time $t' \geqslant t$.

The meaning of $\mathbf{D}_{\mathtt{func}}$ for these rules is:

$$\mathbf{D}^{[\![\cdot]\!]^*}_{\mathtt{func}} = \mathbf{Expr} \to \mathbf{Interp} \to \mathbf{Cache} \to \mathbf{Eqn} \to \Delta \to \mathbf{List}(\mathbb{N}) \to \mathbb{N} \to (\mathbf{D} \times \mathbb{N})$$

$$\begin{aligned}
\mathbf{D}^{[\![\cdot]\!]^*} &= \mathbf{D}_{\mathtt{const}} \cup \mathbf{D}^{[\![\cdot]\!]^*}_{\mathtt{func}} \\
\mathbf{K}^{[\![\cdot]\!]^*} &= (\mathbf{D}_{\mathtt{const}} \cup \mathbf{I}) \to \mathbf{D}^{[\![\cdot]\!]^*}_{\perp} \\
\Delta^{[\![\cdot]\!]^*} &= (\mathbf{D}_{\mathtt{const}} \cup \mathbf{I}) \to \mathbf{D}^{[\![\cdot]\!]^*}_{\perp} \\
\mathbf{Intens}^{[\![\cdot]\!]^*} &= \mathbf{K}^{[\![\cdot]\!]^*} \to \mathbf{D}^{[\![\cdot]\!]^*} \\
\mathbf{Env}^{[\![\cdot]\!]^*} &= \mathbf{Id} \to \mathbf{K}^{[\![\cdot]\!]^*} \to \mathbf{D}^{[\![\cdot]\!]^*}
\end{aligned}$$

We have that $[\![\cdot]\!]^* \in \mathbf{D}^{[\![\cdot]\!]^*}_{\mathtt{func}}$. We will avoid this superscripting when it is not necessary.

Identifiers

x: For the evaluation of x, we need three different kinds of rules.

- The $[\![x]\!]^*\zeta\beta\sigma\delta w t$ rule calls $\langle\!\langle x\rangle\!\rangle^*\zeta\beta\sigma\delta w t\varnothing$, which calculates x with an empty delta.
- $\langle\!\langle x\rangle\!\rangle^*$ iterates through increasing subsets of δ, initially $\varnothing$, until a non-`dem` value is reached. $\langle\!\langle x\rangle\!\rangle^*\zeta\beta\sigma\delta w t\delta'$ calls $\{\!\{x\}\!\}^*\zeta\beta\sigma\delta' w t$. Should a demand for other dimensions be returned, and these are in the domain of δ, then this part of δ is added to δ'.
- The $\{\!\{x\}\!\}^*\zeta\beta\sigma\delta w t$ defines the interaction with the cache.

$$\begin{aligned}
&[\![x]\!]^*\zeta\beta\sigma\delta w t \\
={}& \langle\!\langle x\rangle\!\rangle^*\zeta\beta\sigma\delta w t\varnothing
\end{aligned}$$

$$\begin{aligned}
&\langle\!\langle x\rangle\!\rangle^*\zeta\beta\sigma\delta w t\delta' \\
={}& \text{let } (v, t') = \{\!\{x\}\!\}^*\zeta\beta\sigma\delta' w t \text{ in} \\
&\begin{cases}
(v, t'), & \tau(v) \neq \mathtt{dem} \\
\langle\!\langle x\rangle\!\rangle^*\zeta\beta\sigma\delta w t'\Big(\delta' \dagger \big(\delta \mid \psi(v)\big)\Big), & \psi(v) \subseteq \operatorname{dom}\delta \\
(v, t'), & \text{otherwise}
\end{cases}
\end{aligned}$$

$$\begin{aligned}
&\{\!\{x\}\!\}^*\zeta\beta\sigma\delta w t \\
={}& \text{let } (v, t') = \beta.\text{find}(x, \delta, t, w) \text{ in} \\
&\begin{cases}
(v, t'), & \tau(v) \neq \mathtt{calc} \\
\big(\mathtt{sp}\langle\mathtt{loop}\rangle, t'\big), & v = \mathtt{calc}\langle w'\rangle, w' \geqslant w \\
\{\!\{x\}\!\}^*\zeta\beta\sigma\delta w(t'+1), & v = \mathtt{calc}\langle w'\rangle \\
\quad \text{let } (v', t'') = [\![\sigma(x)]\!]^*\zeta\beta\sigma\delta w t' \text{ in} & \\
\quad \text{let } t''' = \beta.\text{add}(x, \delta, t'', w, v') \text{ in} & \\
(v', t'''), & \text{otherwise}
\end{cases}
\end{aligned}$$

Constants

c: Constants are always constant, and take no time.

$$[\![c]\!]^* \zeta\beta\sigma\delta w t = \big(\zeta(c), t\big)$$

Operations

$op(E_1, \ldots, E_n)$: If one or more operands returns a special, the least special value is returned. If some of the operands return demands, then their union is returned. Let $w_i = cons(i, w)$.

$$\begin{aligned}
&[\![op(E_1, \ldots, E_n)]\!]^* \zeta\beta\sigma\delta w t \\
&= \text{let } (v_i, t_i) = [\![E_i]\!]^* \zeta\beta\sigma\delta w_i t \text{ in} \\
&\begin{cases}
\big(\text{minsp}\{v_i\}, \max\{t_i\}\big), & \exists i.\ \tau(v_i) = \texttt{sp} \\
\Big(\texttt{dem}\big\langle \bigcup_i \{\psi(v_i) \mid \tau(v_i) = \texttt{dem}\}\big\rangle, \max\{t_i\}\Big), & \exists i.\ \tau(v_i) = \texttt{dem} \\
\big(\zeta(op)(v_1, \ldots, v_n), \max\{t_i\}\big), & \text{otherwise}
\end{cases}
\end{aligned}$$

Conditionals

`if` E_1 `then` E_2 `else` E_3 `fi`: Specials and demands from the condition are passed on directly. Otherwise, the correct branch is computed.

$$\begin{aligned}
&[\![\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ fi}]\!]^* \zeta\beta\sigma\delta w t \\
&= \text{let } (v_1, t') = [\![E_1]\!]^* \zeta\beta\sigma\delta w t \text{ in} \\
&\begin{cases}
(v_1, t') & \tau(v_1) = \texttt{sp} \text{ or } \texttt{dem} \\
\big(\texttt{sp}\langle\texttt{type}\rangle, t'\big), & \tau(v_1) \neq \texttt{bool} \\
[\![E_2]\!]^* \zeta\beta\sigma\delta w t', & \psi(v_1) = \texttt{true} \\
[\![E_3]\!]^* \zeta\beta\sigma\delta w t', & \psi(v_1) = \texttt{false}
\end{cases}
\end{aligned}$$

Context query

`#`E: Specials and demands from the contained expression are passed on directly. Should the value not be in the domain of the active delta, then a demand is created for this value. This is the only place where demands are initiated.

$$\begin{aligned}
&[\![\texttt{\#}E]\!]^* \zeta\beta\sigma\delta w t \\
&= \text{let } (v, t') = [\![E]\!]^* \zeta\beta\sigma\delta w t \text{ in} \\
&\begin{cases}
(v, t') & \tau(v) = \texttt{sp} \text{ or } \texttt{dem} \\
\Big(\texttt{dem}\langle v\rangle, t'\Big), & v \notin \text{dom } \delta \\
\big(\delta(v), t'\big), & \text{otherwise}
\end{cases}
\end{aligned}$$

Finite deltas

$[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2}]$: As for data operators, if some of the operands of the tuple operator return demands, then their union is returned. Let $w_{ij} = \mathit{cons}(i * 2 + j, w)$.

$$\begin{aligned}
&[\![[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2}]]\!]^* \zeta\beta\sigma\delta w t \\
&= \text{let } (v_{ij}, t_{ij}) = [\![E_{ij}]\!]^* \zeta\beta\sigma\delta w_{ij} t \text{ in} \\
&\begin{cases}
\big(\text{minsp}\{v_{ij}\}, \max\{t_{ij}\}\big), & \exists i.\ \tau(v_{ij}) = \texttt{sp} \\
\Big(\texttt{dem}\big\langle \bigcup_i \{\psi(v_{ij}) \mid \tau(v_{ij}) = \texttt{dem}\}\big\rangle, \max\{t_{ij}\}\Big), & \exists i.\ \tau(v_{ij}) = \texttt{dem} \\
\big(\texttt{sp}\langle\texttt{tuple}\rangle, \max\{t_{ij}\}\big), & \exists i, k.\ i \neq k \wedge v_{i1} = v_{k1} \\
\big(\texttt{tuple}\langle\{v_{i1} \mapsto v_{i2}\}\rangle, \max\{t_{ij} \mid i = 1..N\}\big), & \text{otherwise}
\end{cases}
\end{aligned}$$

Infinite deltas

$[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2},\ \texttt{all} : E_{\texttt{all}}]$: As for data operators, if some of the operands of the tuple operator return demands, then their union is returned. Let $w_{ij} = \mathit{cons}(i * 2 + j, w)$ and $w_{\texttt{all}} = \mathit{cons}(0, w)$. We use the subscript α instead of ij or $\texttt{all}$.

$$\begin{aligned}
&[\![[E_{11} : E_{12},\ \ldots,\ E_{n1} : E_{n2},\ \texttt{all} : E_{\texttt{all}}]]\!]^* \zeta\beta\sigma\delta w t \\
&= \text{let } (v_\alpha, t_\alpha) = [\![E_\alpha]\!]^* \zeta\beta\sigma\delta w_\alpha t \text{ in} \\
&\begin{cases}
\big(\text{minsp}\{v_\alpha\}, \max\{t_\alpha\}\big), & \exists i.\ \tau(v_\alpha) = \texttt{sp} \\
\Big(\texttt{dem}\big\langle \bigcup_i \{\psi(v_\alpha) \mid \tau(v_\alpha) = \texttt{dem}\}\big\rangle, \max\{t_\alpha\}\Big), & \exists i.\ \tau(v_\alpha) = \texttt{dem} \\
\big(\texttt{sp}\langle\texttt{tuple}\rangle, \max\{t_\alpha\}\big), & \exists i, k.\ i \neq k \wedge v_{i1} = v_{k1} \\
\left(\begin{array}{c} \texttt{tuple}\langle\{v \mapsto v_{\texttt{all}} \mid v \in (\mathbf{D}_{\texttt{const}} \cup \mathbf{I})\} \dagger \\ \{v_{i1} \mapsto v_{i2} \mid i = 1..N\}\rangle, \end{array} , \max\{t_\alpha\}\right), & \text{otherwise}
\end{cases}
\end{aligned}$$

Context perturbation

$E_2\ \texttt{@}\ E_1$: The rule for context-change operators does not change.

$$\begin{aligned}
&[\![E_2\ \texttt{@}\ E_1]\!]^* \zeta\beta\sigma\delta w t \\
&= \text{let } (v_1, t') = [\![E_1]\!]^* \zeta\beta\sigma\delta w t \text{ in} \\
&\begin{cases}
(v_1, t') & \tau(v_1) = \texttt{sp} \text{ or } \texttt{dem} \\
\big(\texttt{sp}\langle\texttt{type}\rangle, t'\big), & \tau(v_1) \neq \texttt{tuple} \\
[\![E_2]\!]\zeta\beta\sigma\big(\delta \dagger \psi(v_1)\big) w t', & \text{otherwise}
\end{cases}
\end{aligned}$$

Value-parameter application

$(\lambda_\ell\, E)$: We must take into account the time parameter.

$$\begin{aligned}
[\![(\lambda_\ell\, E)]\!]^* \zeta\beta\sigma\delta w t = {} & \lambda E'.\lambda\zeta'.\lambda\beta'.\lambda\sigma'.\lambda\delta'.\lambda w'.\lambda t' \\
& \text{let } \sigma'' = \sigma \cup \{\chi_\ell \mid \mathit{true} \to \texttt{\#}\gamma_\ell\} \text{ in} \\
& \text{let } (v', t'') = [\![E']\!]^* \zeta'\beta'\sigma'\delta' w' t' \text{ in} \\
& \text{let } \delta'' = \delta' \dagger \{\gamma_\ell \mapsto v'\} \text{ in} \\
& [\![E]\!]^* \zeta\beta\sigma''\delta'' w t''
\end{aligned}$$

$$\begin{aligned}
[\![E_1.E_2]\!]^* \zeta\beta\sigma\delta w t = {} & \text{let } (v, t') = [\![E_1]\!]^* \zeta\beta\sigma\delta w t \text{ in} \\
& [\![v(E_2)]\!]^* \zeta\beta\sigma\delta w t'
\end{aligned}$$

Named-parameter application

$(\phi_\ell\, E)$: We must take into account the time parameter. We assume that E_1 has been tagged with value j.

$$\begin{aligned}
\left[\!\left[(\phi_\ell\, E)\right]\!\right]^* \zeta\beta\sigma\delta w t = {} & \lambda E'.\lambda\zeta'.\lambda\beta'.\lambda\sigma'.\lambda\delta'.\lambda w'.\lambda t' \\
& \text{let } \sigma'' = \sigma \cup \left\{\chi_\ell \mid hd(\#\gamma_\ell) \equiv j \to E' \mathbin{@} [\gamma_\ell : tl(\#\gamma_\ell)]\right\} \text{ in} \\
& \text{let } \delta'' = \delta' \dagger \left\{\gamma_\ell \mapsto cons(j, \delta'(\gamma_\ell))\right\} \text{ in} \\
& [\![E]\!]^* \zeta\beta\sigma''\delta'' w t'
\end{aligned}$$

$$\begin{aligned}
[\![E_1\ E_2]\!]^* \zeta\beta\sigma\delta w t = {} & \text{let } (v, t') = [\![E_1]\!]^* \zeta\beta\sigma\delta w t \text{ in} \\
& \left[\!\left[v(E_2)\right]\!\right]^* \zeta\beta\sigma\delta w t'
\end{aligned}$$

Local definitions

The local definitions are handled by augmenting the set of equations, renaming the variables to avoid any name clashes.

$$\begin{aligned}
& [\![E \ \texttt{where}_\ell\ \gamma_{\ell,i=1..m}\ q_{j=1..n}]\!]^* \zeta\beta\sigma\delta w t \\
= {} & \text{let } \delta' = \delta \dagger \{\gamma_{\ell,i} \mapsto init \mid i = 1..m\} \text{ in} \\
& \text{let } \sigma' = \sigma \cup \{q_j \mid j = 1..n] \text{ in} \\
& [\![E]\!]^* \zeta\beta\sigma'\delta' w t
\end{aligned}$$

Equations and bestfitting

$$\begin{aligned}
[\![q]\!]^* \zeta\beta\sigma\delta w t = {} & \text{let } \{g_1, \ldots, g_n\} = \ \big\{g \in q \mid [\![B_g]\!]^* \zeta\beta\sigma\delta w t \equiv (true, t_g) \wedge \\
& \qquad\qquad\qquad\qquad (\nexists g' \in q)\ moreSpecific(B_{g'}, B_g)\big\} \text{ in} \\
& \text{let } t' = \max\{t_g\}_{g \in q} \text{ in} \\
& \text{let } (v_i, t_i) = [\![E_{g_i}]\!]^* \zeta\beta\sigma\delta w t' \text{ in} \\
& \big(v_1 \oplus \cdots \oplus v_n, \max\{t_i\}_{i=1..n}\big), \qquad n > 0 \\
& (\texttt{sp}\langle\texttt{undef}\rangle, t'), \qquad\quad \text{otherwise}
\end{aligned}$$

where $moreSpecific(B_{g'}, B_g)$ means that the region $\{\kappa \mid [\![B_{g'}]\!]^* \zeta\beta\sigma\delta w t \equiv (true, t_{g'})\}$ is *provably* (computationally) *strictly smaller* than the region $\{\kappa \mid [\![B_g]\!]^* \zeta\beta\sigma\delta w t \equiv (true, t_g)\}$.

4.6.4 Results

Proposition 11. *Let ξ be a function-free environment and κ be a function-free context. Suppose furthermore that $\forall x.\forall\kappa.\xi(x)(\kappa) = \beta^*(x, \kappa, 0, nil)$, and that $v \in \mathbf{D}_{\texttt{const}}$. Then $\langle\!\langle E\rangle\!\rangle\xi\sigma\kappa = v$ iff $[\![E]\!]^* \beta\sigma\delta(nil)0 = (v, t)$, for some $\delta \sqsubseteq \kappa$, and $t \geqslant 0$.*

The restriction on ξ is to ensure that it can be considered to be an element both of $\mathbf{Env}$ and $\mathbf{Env}^{[\![\cdot]\!]^*}$. Similarly for κ, $\mathbf{K}$ and $\mathbf{K}^{[\![\cdot]\!]^*}$. The proof is a straightforward double induction, first on the structure of the expressions, second on the interaction between the evaluator and the cache.

4.7 Conclusions

The key to the development of the semantics in this chapter is the proper definition of context and of context delta. Once these were defined, the rules for expressions not involving identifiers, nor functions, wrote themselves.

The identifiers were trickier. The choice of an abstract syntax sufficiently simple, yet general enough to cover many possible concrete syntaxes, required some thought. As for bestfitting, initial work, as described in the chapter on possible-worlds versioning, assumed that there would always be a single bestfit equation. The solution given here allows for multiple bestfits, all of which are then evaluated and combined *post facto*. This solution, much more general, is also much simpler.

Initially, all functions were perceived as having two different kinds of arguments. However, by trying to make Curry-ed functions, it became clear that there were really two kinds of abstraction, hence the use of λ and ϕ. The naming of the two different kinds of application came from their calling mechanism.

Alternative semantics have also been examined, but are not presented here. First, by using an *closure type*, it is possible to make the delta evaluation lazy. In this approach, the dimensions of a delta must be computed eagerly upon creation, while the ordinates are wrapped up in closures including the current context. An ordinate closure is only opened and evaluated if the ordinate is requested through the use of a `#`. For computations that yield a non-$\perp$ value using the rules given in this chapter, the result is the same. However, there could be situations where the evaluation of unused components of deltas might be nonterminating, in which case the two solutions would yield different results.

In this thesis, we will consider lazy tuples to be an implementation—*not* semantic—feature, useful for optimization. This was the approach taken by Toby Rahilly, when he developed the first multi-threaded implementation of the language [80].

Another alternative semantics that has been written is to consider all values to be sets. The intuition came from a 1977 paper by Adi Shamir and William Wadge [85]. In this paper, they proposed a theory of types, in which types and values are the same, and a type is understood as the set of all values approximating it. In this view, no object has a canonical type. We are currently experimenting with this idea in the current implementation, for the bestfit selection of operators based on the types of arguments.

The key to the successful semantics was the recognition that there are two kinds of parameter passing and two kinds of binding. Value-parameter abstraction uses call-by-value while named-parameter abstraction uses call-by-name. Binding of identifiers in the environment is done statically, while binding of dimensions in the context is done dynamically. All are useful and necessary.

Chapter 5

Building an Interpreter

In this chapter, we present the interpreter that implements the TransLucid language [8]. The main difference between the Core TransLucid language presented in the previous chapters and the language whose implementation is described here is the richness and diversity of data types and operations. In particular, the TransLucid presented here is designed as a *coordination language*, which means that it provides an interface to other structures written in a host language.

The TransLucid programming environment, available at `translucid.sourceforge.net`, is written in `C++0x`, the new standard for `C++`. For compilation, it currently requires GNU `g++ 4.5.0` (`gcc.gnu.org`), and the Boost `1.43.0 C++` libraries (`www.boost.org`).

In addition to the three atomic data types described in the previous chapter, the environment natively supports the `intmp` (GNU multi-precision integer), `uchar` (32-bit Unicode character) and `ustring` (32-bit Unicode string) data types. In addition, users may add other data types and operations through the use of libraries, and these may be manipulated from TransLucid. Furthermore, the highly flexible parser can be parameterized so that new operators can be added, in prefix, postfix or infix form, and literals (constants) of new types can be written as simply as if they were literals of the builtin types.

The interpreter is designed to allow the interaction of the host language, here `C++`, with TransLucid: TransLucid can call TransLucid or `C++`, and `C++` can call `C++` or TransLucid. The means by which this communication takes place is the *hyperdaton*, an object which responds to a context (a *tuple*) and returns a *tagged value*, a (*constant*, *tuple*) pair, where the tuple encodes the subcontext used to compute the constant.

The hyperdaton is the key data structure. A hyperdaton can be generated by hand in `C++` or else can be produced automatically from TransLucid equations. All forms of expression are subclasses of the hyperdaton class, as are variables, equations, and the system itself.

The *variable* is a specialization of hyperdaton, used to store all of the information pertaining to a variable, including its defining equations, and possibly variables local to it.

The *system* is a specialization of variable, used to store a number of variables, along with a set of input hyperdatons used by the equations defining the variables, and a set of output hyperdatons, which are filled in through the use of *demands*, essentially reservations for computations to take place at specific contexts.

The semantics of a system is synchronous. Upon each instant, the input hyperdatons, equations and demands may be updated. The applicable demands are then executed, in so doing updating the output hyperdatons. This process is repeated upon each instant.

Since the Cartesian system is to be used as a real system, we must be able to deal with all of the difficulties of user-defined types, of libraries, of identifier dimensions, and so on. As a result, the internals of a system are designed so that they can be understood from a TransLucid perspective: this is done through the use of predefined dimensions and variables, so that, for example, adding a new operator into the system effectively means adding a new equation into the system.

The presentation style of this chapter is ground up. As the internals of the `C++` system are explained, the corresponding TransLucid structures will be presented.

5.1 Types

Inside the interpreter, each type is given a unique type index, an unsigned integer (`type_index`).

The following types are predefined and necessary for the behavior of the system:

- `type`: types (`TYPE_INDEX_TYPE`). The values are indices corresponding to registered types.
- `dim`: dimensions (`TYPE_INDEX_DIMENSION`). The values are indices corresponding to dimensions.
- `sp`: specials (exceptional values, `TYPE_INDEX_SPECIAL`). The values are indices corresponding to registered special values.
- `tuple` is used to hold tuples (`TYPE_INDEX_TUPLE`).
- `guard` is used to hold guards (`TYPE_INDEX_GUARD`).
- `range` is used to hold ranges of integers (`TYPE_INDEX_RANGE`).

The following types will soon be added to the interpreter.

- `calc` is used to hold a reference to a worker thread undertaking an unfinished computation.
- `expr` is used to hold expressions.
- `eqn` is used to hold equations.

These are the four basic types that are found in most programming languages and which are also necessary for the proper behavior of the system:

- `bool`: Booleans (`TYPE_INDEX_BOOL`).
- `intmp`: GNU MP arbitrary-precision integers (`http://gmplib.org`, `TYPE_INDEX_INTMP`).
- `uchar`: 32-bit Unicode character (`http://www.unicode.org`, `TYPE_INDEX_UCHAR`). The range is `0`–`D7FF`, `E000`–`10FFFF` hex.
- `ustring`: 32-bit Unicode character string (`TYPE_INDEX_USTRING`).

In addition to these predefined types, it is possible to add new user types through the use of libraries.

5.2 Typed values

When evaluating expressions, the TransLucid interpreter can manipulate values of both builtin types as well as user-defined types. In order for the interpreter to work properly, it must hold on to *typed values*, which are values whose type is known. All of these values are instances of subclasses of class `TypedValue`.

```
class TypedValue
{
  virtual ~TypedValue() {}
  virtual TypedValue* clone() const = 0;
  virtual size_t hash() const = 0;
  virtual void print(ostream& os) const = 0;
};
```

Depending on the type, these subclasses might simply store the values directly, as appropriate for small fixed-size types; or using a pointer to a possibly shared data structure, as needed for larger variable-size types.

5.3 Constants

Inside the interpreter, constants are pairs consisting of a typed value and the type index for the constant's type. The type is called `Constant`, and its main constructor is as follows:

```
Constant(const TypedValue& value, type_index index);
```

In expressions, constants in general appear in expressions as *typename*`<`*valuetext*`>`, where *typename* is a known type and *valuetext* is the text that is to be interpreted by that type's own parser.

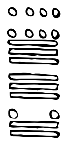
The four basic types can appear directly in expressions.

- The values for the `sp` type are:
  - `sperror`: General-purpose error.
  - `spaccess`: Change to unreachable context.
  - `sptype`: Type error.
  - `sptuple`: Dimensions for tuple formation are not disjoint.
  - `spdim`: Undefined dimension in the current context.
  - `spundef`: Undefined variable.
  - `spmultidef`: Multiply defined variable.
  - `spconst`: Error when parsing a constant.
  - `sploop`: A value is determined in terms of itself.
  - `spmultival`: A single value is expected.
- The two possible Boolean values are `true` and `false`.
- Integers beginning with a non-zero digit are understood to be decimal numbers. Their syntax is `[1-9][0-9]*`.
- Zero is expressed simply as a standalone zero digit.
- Other integers beginning with a zero digit are expressed in a base between 2 and 61. Their syntax is `0[2-9a-zA-Z][0-9a-zA-Z]*`. The digit or letter following the initial digit gives the base, and what follows is the number. The digits and letters are interpreted as follows:

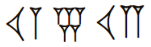
`0`	`1`	`2`	`3`	`4`	`5`	`6`	`7`	`8`	`9`	`a`	`b`	`c`	`d`	`e`	`f`
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`g`	`h`	`i`	`j`	`k`	`l`	`m`	`n`	`o`	`p`	`q`	`r`	`s`	`t`	`u`	`v`
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
`w`	`x`	`y`	`z`	`A`	`B`	`C`	`D`	`E`	`F`	`G`	`H`	`I`	`J`	`K`	`L`
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
`M`	`N`	`O`	`P`	`Q`	`R`	`S`	`T`	`U`	`V`	`W`	`X`	`Y`	`Z`		
48	49	50	51	52	53	54	55	56	57	58	59	60	61		

  For a number in base n, only 'digits' from 0 to $n-1$ may be used. In computer science, the most commonly used bases are 2 (binary), 8 (octal) and 16 (hexadecimal); in everyday life, we use base 10 (decimal). Consider the number 39912. It becomes
  - `021001101111101000` in binary (base 2).
  - `08115750` in octal (base 8).
  - `0a39912` in decimal (base 10).
  - `0g9be8` in hexadecimal (base 16).

  - `0k4jfc` in vigesimal (base 20), used by the Mayans.



  - `0Yb5c` in sexagesimal (base 60), used by the Babylonians.



- A single Unicode character in UTF-8 format enclosed in single quotes is understood as a `uchar`.
- A sequence of Unicode characters in UTF-8 format enclosed in double quotes is understood as a `ustring`.

When a new type is introduced, a parser and a pretty-printer for that type must be provided as well.

5.4 Dimensions

Inside the interpreter, each dimension is given a unique dimension index, an unsigned integer (`dim_index`). Dimensions can either be typed values or identifiers declared to be dimensions.

The following dimensions are predefined and necessary for the behavior of the system:

- `id` (`DIM_ID`): variable inside a hyperdaton.
- `name` (`DIM_NAME`): name of operation.
- `text` (`DIM_TEXT`): value text for a constant.
- `type` (`DIM_TYPE`): type string for a constant.
- `_validguard` (`DIM_VALIDGUARD`): guard of an equation.
- `value` (`DIM_VALUE`): value of a value dimension.
- `time` (`DIM_TIME`): current time.
- `arg`i, $i \geqslant 0$ (`DIM_ARG`i): i-th argument of an operation.
- `all` (`DIM_ALL`): the default value for missing dimensions.

5.5 Tuples

Tuples are used to encode the current context. For the moment, only eager tuples are implemented. Physical tuples are mappings from dimension indexes to constants. The `Tuple` class holds a shared pointer to a physical tuple: many `Tuple` instances can share the same physical tuple, and garbage collection is easier.

```
typedef map<dim_index, Constant> tuple_t;

class Tuple : public TypedValue
{
  explicit Tuple(const tuple_t& tuple);
  Tuple();

  private:
    boost::shared_ptr<tuple_t> m_value;
};
```

5.6 Tagged constants

The caching mechanisms described in Chapter 4 need to know exactly what part of the context was used to calculate the current value. As a result, calculations result in a `TaggedConstant`, a pair including a constant and a tuple.

```
typedef pair<Constant, Tuple> TaggedConstant;
```

5.7 Parsing expressions

The Core TransLucid language simply assumes that there is a set of constants and operators. However, the real language is to be a coordination language, allowing the use of constants and operators defined in the host language, here `C++`. Furthermore, we would like to allow a flexible, extensible concrete syntax, with the use of user-defined operators.

To do this, a header is used to initialize the expression parser, in order to interpret tokens appearing in expressions. This allows data operators, identifiers and constants to be properly interpreted.

There are no predefined data operators in TransLucid. These must all be declared in the header. They may be unary (prefix or postfix) or infix. The infix operators may be binary or variadic (2 or more arguments). Each operator, once recognized during the parsing process, is converted to a function name for internal processing. The fixity declarations are as follows:

- `postfix` s_1 s_2: declares s_1 to be a postfix operator, which translates to internal function s_2 upon being parsed. Postfix operators have highest priority. (All s occurrences are strings.)
- `prefix` s_1 s_2: declares s_1 to be a prefix operator, which translates to internal function s_2 upon being parsed. Prefix operators have next highest priority.
- The following all declare s_1 to be an infix operator, which translates to internal function s_2 upon being parsed, and is of precedence level n (higher precedence means higher n). Unary operators have higher precedence than infix operators.
  - `infixl` s_1 s_2 n: declares s_1 to be a left-associative binary operator.
  - `infixr` s_1 s_2 n: declares s_1 to be a right-associative binary operator.
  - `infixn` s_1 s_2 n: declares s_1 to be a non-associative binary operator.
  - `infixm` s_1 s_2 n: declares s_1 to be a variadic ordinary operator. For example, $2 + 3 + 4$ can be considered to be $+(2, 3, 4)$.
  - `infixp` s_1 s_2 n: declares s_1 to be a variadic comparison operator. For example, $2 < 3 < 4$ can be considered to be $2 < 3 \wedge 3 < 4$.

Identifiers in TransLucid can either be variables or dimensions. To distinguish the two, dimension identifiers must be declared.

- `dimension` s: declares s to be a dimension when it appears in expressions.

Constants of user-defined types can be placed in expressions of the form *typename*`<`*valuetext*`>`. However, if *typename* is used often, then it is possible to define a shorthand, using *user-defined delimiters*.

- `delimiters` s c_1 c_2: declares that a string s appearing between characters c_1 and c_2 in an expression will be considered to be equivalent to *typename*`<`*s*`>`.

Types and functions implementing data operators can be defined in the host language and placed in libraries, which can then be loaded by TransLucid.

- `library` s: declares s to be a dynamic library which must be loaded.

Libraries are set up to resemble sets of equations, using predefined dimensions.

5.8 Hyperdatons

At the core of the implementation is the *hyperdaton*, an arbitrary-dimensional unbounded array. The C++ class HD is given below:

```
class HD
{
  virtual ~HD() {}
  virtual TaggedConstant operator()(const Tuple& k) = 0;
  virtual uuid addExpr(const Tuple& k, HD* h)
  {
    return nil_uuid();
  }
}
```

In C++, an object x instantiating a class C with the method operator() method defined is called a *functor*, or *function object*: when one writes x(*actuals*), if operator()(*formals*) is defined for C and of the right type, then it is called; should there be more than one operator()(*formals*) defined for C, i.e., operator() is *overloaded*, then the one with the most specific type for the *actuals* is called.

The operator() method is used to do a lookup in the current context, which is encoded using the data structure Tuple. This lookup might simply require, internally, the return of a constant or a table lookup. Or, it might require some massive computation. The returned result, of type TaggedConstant, contains the constant, and the pertinent subcontext (of the original context passed in as input) that was actually needed to compute the result.

The interpreter consists therefore of a number of classes defining different kinds of hyperdaton, one for each of the different kinds of expression, along with a parser that converts expressions and equations into hyperdatons and a pretty-printer that allows printing of tagged values and hyperdatons. As a result, it is possible to do Cartesian programming either in C++ or in TransLucid, or through a judicious mix of the two.

The addExpr method is used to add equations used by a hyperdaton to change its behavior. The uuid type is a *universally unique identifier* generated by the Boost.Uuid library; it is a 128-bit identifier that is guaranteed—or at least extremely likely—to be unique, even in a distributed environment. Using this identifier, it becomes possible to replace or delete an expression that has been added to a hyperdaton.

5.9 Expression hyperdatons

For the expression hyperdatons presented below, some of them will require having access to the current *system* in which they are defined. This is because, internally, many parts of a system are encoded using bestfitting, through the use of special dimensions and variables.

When ones write x as an identifier, the TransLucid interpreter understands this as

```
ID @ [name: x]
```

Therefore, all variables are understood to be members of the same hyperdaton, namely ID.

Similarly, when one writes $3 + 5$, the TransLucid interpreter will look up the function name, in this case operator+, and encode the whole thing as

```
OP @ [name:"operator+", arg0:3, arg1:5]
```

Therefore, all operators are understood to be members of the same hyperdaton, namely OP. This approach is used for loading of libraries: when a library is loaded, it is passed the current system; the library then adds to that system equations of the following form:

```
OP @ [name:"operator+", arg0:type<intmp>, arg1:type<intmp>]
   = <Boost.Function object>
```

where a `Boost.Function` is a class which can encapsulate either a function or a function object, thereby providing a unified interface. As a result, when `operator+` is to be evaluated, the best-fitting process can be used to determine which instance of `operator+` is to be used. A similar process is used to keep track of types, dimensions and demands. In time, even the parser and the pretty-printer will be defined this way.

5.9.1 Constant hyperdaton

The constant hyperdatons hold a single constant, and no matter what the context, will always return the same value. There is one for all the builtin data types that are writable in TransLucid.

```
TypeConstHD (type_index);
BoolConstHD (bool);
SpecialConstHD (Special::Value);
IntmpConstHD (const mpz_class&);
UCharConstHD (char32_t);
UStringConstHD (const u32string&);
```

5.9.2 Typed-value hyperdaton

Expressions of the form *typename*`<`*valuetext*`>` are "constants", yet they are not. The TransLucid interpreter will call the parser for *typename* on the string *valuetext*, and this parser is context-dependent. The hyperdaton storing such expressions is the `TypedValueHD`.

```
TypedValueHD (HD* system, const u32string& type, const u32string& text);
```

5.9.3 Dimension hyperdaton

In expressions, `DimensionHD` is used for identifiers that have been declared to be dimensions.

```
DimensionHD (HD* system, const u32string& name);
```

5.9.4 Identifier hyperdaton

In expressions, `IdentHD` is used for identifiers defined by systems of equations.

```
IdentHD (HD* system, const u32string& name);
```

5.9.5 Operator hyperdaton

The abstract syntax referred to "operators". Internally, there are hyperdatons for unary operators, binary operators, and `isspecial` and `istype`.

```
UnaryOpHD    (HD* system, const u32string& name, HD* e);
BinaryOpHD   (HD* system, const u32string& name, const vector<HD*>& operands);
VariableOpHD (HD* system, const u32string& name, const vector<HD*>& operands);
IsSpecialHD  (HD* system, const u32string& special, HD* e);
IsTypeHD     (HD* system, const u32string& type, HD* e);
```

In TransLucid, `isspecial`$\langle$*name*$\rangle$ E always returns a Boolean value; it will evaluate E and determine if it is `sp`$\langle$*name*$\rangle$. Similarly, `istype`$\langle$*name*$\rangle$ E always returns a Boolean value; it will evaluate E and determine if it is of type *name*. Therefore, `isspecial` and `istype` are not strict with respect to specials.

5.9.6 Conditional hyperdaton

Conditional, context query, tuple creation and context perturbation expressions are all handled in a straightforward manner:

```
IfHD (HD* cond, HD* then, const vector<pair<HD*, HD*>>& elsifs, HD* else_);
HashHD (HD* system, HD* e);
TupleHD (HD* system, const list<pair<HD*, HD*>>& elements);
AtHD (HD* e2, HD* e1);
```

5.10 Equation hyperdatons

Equations are also hyperdatons. The variable *name* is being defined. The guard *valid* defines the validity conditions of equations. The hyperdaton h is the defining expression.

```
EquationHD(const u32string& name, const GuardHD& valid, HD* h);
```

5.11 Variable hyperdatons

The `VariableHD` hyperdaton is given below:

```
typedef std::map<uuid, VariableHD*> UUIDVarMap;
typedef std::map<uuid, EquationHD> UUIDEquationMap;
typedef std::map<u32string, VariableHD*> VariableMap;

class VariableHD: public HD
{
  uuid addExpr (const Tuple& k, HD* h);
  void delExpr (const uuid);
  void replExpr(const uuid, const Tuple& k, HD* h);

  ...

  private:
    UUIDVarMap m_uuidVars;
    UUIDEquationMap m_equations;
    VariableMap m_variables;
};
```

The `addExpr` method is used to add a new definition, itself encoded as a hyperdaton, to the variable hyperdaton. A single variable may have multiple definitions, and when a lookup takes place, the most specific, with respect to the current context, will be used. The `uuid` returned as result from `addExpr` corresponds to a unique identifier generated internally so that the definition may be referred to explicitly. The `delExpr` and `replExpr` are used to, respectively, delete and replace an existing equation. The `m_equations` field holds on to the current equations for this variable.

The `VariableHD` hyperdaton can also hold a number of variables lower down, so that one can write dot-separated expressions. For example, $x.y.z$ means the z variable in the y variable in the x variable.

The `m_uuidVars` field maps `uuid`'s to children variables. The `m_equations` field holds on to the equations for the current variable. The `m_variables` field holds on to the children variables.

The following variables are predefined and necessary for the behavior of the system:

- `CONST`: constants.

  - `type`: type name of the constant (a string).
  - `text`: text of constant, to be parsed in a context-dependent way.

- `OP`: operators.
  - `name`: name of the operator.
  - `arg`i, $i \geqslant 0$: arguments of the operator.

- `TYPE_INDEX`: type indices.
  - `type`: type name (a string).

- `DIMENSION_NAMED_INDEX`: dimension indices.
  - `name`: name of the dimension.

- `DIMENSION_VALUE_INDEX`: dimension indices.
  - `value`: value designating the dimension.

5.12 System hyperdatons

The `SystemHD` hyperdaton is derived from `VariableHD`:

```
class SystemHD: public VariableHD
{
  void addInput (const map<u32string, HD*>);
  void addOutput(const map<u32string, HD*>);
  void addDemand(const u32string& id, const EquationGuard& guard);
  void tick();
};
```

It includes equations and is passed *physical* hyperdatons as input and output. In this way, it is possible to have hyperdatons that have their own `C++`-specific interface for use outside of TransLucid, as well as the hyperdaton interface for use within TransLucid. The demands are requests for output hyperdatons to be calculated under certain contexts.

Once all of the equations and demands are added, then the `tick()` method can be called, producing all of the results, and the demands are executed, with the results being placed in the output hyperdatons.

In some sense, everything is a hyperdaton. This approach has simplified the programming of the interpreter. In particular, identifiers and operators are encoded using hyperdatons, through the use of special identifiers.

5.13 Current work

At the time of writing, the current implementation does not correspond directly to the Core TransLucid presented in Chapter 4. There are currently no functions, nor local `where` clauses. However, the infrastructure to implement these is already included in the implementation.

To implement value-parameter functions simply requires manipulation of the parser. To implement named-parameter functions, however, requires the ability to manipulate finite lists. This will be implemented in TransLucid by allowing the creation of inductively defined data structures, as are available in most functional languages. In TransLucid, these data structures will be implemented as tuples (already implemented), with two predefined dimensions: `type` will determine the type and `constructor` will determine which kind of object of that type will be created. Once these types will be available, then implementing functions in TransLucid will be doable.

As for the local definitions, the current implementation already allows local definitions, since every variable allows the use of local variables. These structures can be used to implement the `where` clauses of Core TransLucid.

Efficiency considerations are currently under study. Several implementations of earlier versions of TransLucid have been built with caches, including for the minimal cache described in Chapter 4. With this knowledge, we will develop a cache for the current implementation, with programmer support for garbage-collection strategies.

We are also examining means for highly efficient implementations of regularly occurring structures. See Chapter 7 for a discussion about the implementation of recursive functions.

5.14 Conclusions

The interpreter presented here does much more than simply implement the Core TransLucid language. It is, in fact, designed as the bootstrapping step towards an interpreter for TransLucid that will be entirely written in TransLucid.

The key concept in the interpreter is the hyperdaton, which provides a result given a context. By subclassing the hyperdaton, and by using a restricted set of predefined dimensions, much of the behavior of the interpreter comes for free through the bestfitting process.

The development of Cartesian programs combining the full power of TransLucid and of the host language will inevitably lead to new programming patterns. We look forward to this experimentation.

Part III

Exploring the Cartesian Space

Chapter 6

Time for Applications

In this chapter, we present two TransLucid-based applications, `tlcore` and `S`^{3}, and examine how time is used in these applications and in distributed synchronous systems.

The `tlcore` application is a standalone text-based TransLucid interpreter, with a synchronous reactive semantics. At each instant, one may add, replace or delete defining equations and expressions to be evaluated, and the special `time` dimension may be referred to in order to keep track of the evolution of the system of equations.

The `S`^{3} application is a standalone graphical TransLucid code and expression browser, in which one can edit the set of defining equations and expressions to be evaluated, and in which can visualize the evaluation of these expressions in multidimensional space. Because of the exploratory nature of the tool, time may advance in micro-steps, whenever the equations or expressions are modified, or in macro-steps, when a commit to the changes is made.

These standalone tools have allowed the development of the timed semantics of the TransLucid system in the library presented in the previous chapter. Using this semantics, we can then explore different ways of having multiple TransLucid systems interacting. We present three models: the peer-to-peer model, the hierarchical-worker model and the parent-æther model.

One of the longer-term goals of TransLucid is that it can be used for the writing of timed systems. In the presentation in this chapter, we focus on allowing a system of TransLucid equations to be reactive, i.e., that the set of equations available may differ from instant to instant.

6.1 The `tlcore` application

The `tlcore` application is a simple standalone application with an interactive loop to interpret and evaluate TransLucid programs. It is used both to test developments within the TransLucid library as well as to allow users to directly write and run TransLucid programs. `tlcore` forms part of the TransLucid distribution.

6.1.1 Basic behavior

In its simplest form, `tlcore` reads either from a file or from the standard input: a TransLucid header, a set of equations and a set of expressions. The expressions are then evaluated one by one, and the results are placed on the standard output or in a file. The interface is as follows:

`tlcore --input=`*infile* `--output=`*outfile*

along with the standard `--verbose` and `--version` options.

If the `--input` option is not provided, input is read from the standard input. If the `--output` option is not provided, output is written to the standard output. All errors or log messages, including version information, are written to the standard error.

The syntax of the files is simple. It consists of three parts:

header
%%
equations
%%
expressions

where

- *header* consists of declarations for libraries, dimensions, delimiters and operators (see §5.7).
- *equations* consists of equations, all terminated by double colons, using the information declared in the header.
- *expressions* consists of expressions, all terminated by double colons, using variables defined in the equations.

The expressions are to be read, one by one, and then evaluated. Suppose that there are n expressions. Their answers will be returned as follows:

$answer_1$; ;
. . .
$answer_n$; ;

Should the `--verbose` option be used, then the output will be

$expr_1 \rightarrow answer_1$; ;
. . . ; ;
$expr_n \rightarrow answer_n$; ;

6.1.2 Equation UUIDs

When an equation is added to the TransLucid system, it is automatically given a unique identifier, generated by the Boost.UUID library. When `tlcore` is passed the `--uuid` option, then both the equations and the expressions will return results. The results will be of the form

// Equations
$eqnuuid_1$; ;
. . .
$eqnuuid_m$; ;

// Expressions
$answer_1$; ;
. . .
$answer_n$; ;

Should the `--verbose` option also be set, then the results will be of the form

// Equations
$eqn_1 \rightarrow eqnuuid_1$; ;
. . .
$eqn_m \rightarrow eqnuuid_m$; ;

// Expressions
$expr_1 \rightarrow answer_1$; ;
. . .
$expr_n \rightarrow answer_n$; ;

Here is a sample run from `tlcore --uuid`:

```
%%
x = 5;;
%%
x ;;

// Equations
uuid<4557f67592cb498fa684d50f5511a65> ;;

// Expressions
intmp<5> ;;
```

6.1.3 Reactive `tlcore`

When the `--reactive` option is set for `tlcore`, then the application becomes reactive, and the set of equations is allowed to vary over time, and the expressions being evaluated can refer to the `time` dimension. The idea is that at each instant,

- the header may be added to;
- the set of equations may be modified;
- the set of expressions to be evaluated may be modified; and
- then the expressions for that instant are evaluated.

When `tlcore` is reading from the standard input, then the input will be of the form

```
header_0
%%
equations_0
%%
expressions_0
$$

header_1
%%
equations_1
%%
expressions_1
$$

...
```

Each instant corresponds to the lines appearing between successive occurrences of `$$`. The output is of the same form as the input, with occurrences of `$$` to separate the different instants.

When the input is given in the form of option `--input=`*infile*, then the entire input, for all instants, is read from *infile*. When the input is given in the form of option `--inputiter=`*inprefix*, then the input for instant n comes from file $inprefix_n$. In other words, the input for each instant is placed in a different file. Similarly for `--output` and `--outputiter`.

If the `--uuid` option is set, then the universal identifiers for the equations are presented to the programmer, who can then manipulate them explicitly, with lines of the form:

```
delete eqnuuid_1 ;;
replace eqnuuid_2 eqn ;;
```

For the first line, if the current value of `#time` is t, then from t on, $\mathit{eqnuuid}_1$ is no longer usable. For the second line, the equation corresponding to $\mathit{eqnuuid}_2$ is replaced, from t on, with *eqn*. If the value of `#time` is changed to $t' < t$, then all of the equations available at time t' continue to be accessible.

Changes to the set of equations are done synchronously. All additions, deletions and replacements for a given instant are done as a single transaction, and must be consistent.

An empty instant stops the `tlcore` program.

6.1.4 Adding demands

When the `--demands` option is set in `tlcore`, then it can handle *demands*, which are requests for (*identifier*, *context*) pairs to be evaluated, should this be possible. Demands are necessary for the development of large-scale Cartesian programs using the `C++` interface, to force `C++` data structures to be updated through the TransLucid interpreter.

When demands are used, the TransLucid instants have four parts, not three. The fourth part is introduced with another `%%` pair, as follows:

header_0
`%%`
$\mathit{equations}_0$
`%%`
$\mathit{expressions}_0$
`%%`
$\mathit{demands}_0$
`$$`

header_1
`%%`
$\mathit{equations}_1$
`%%`
$\mathit{expressions}_1$
`%%`
$\mathit{demands}_1$
`$$`

...

A demand requests that a variable be computed in a specific context. It is written as a pair

$$(x, \mathit{tuple})\ ;;$$

and it is registered internally by the system with a UUID. At each instant, should the tuple be valid in that instant, then this pair is evaluated, producing the result, along with the other expressions. The response for an instant therefore looks like this:

// Expressions
$\mathit{answer}_1\ ;;$
...
$\mathit{answer}_n\ ;;$

// Demands
$\mathit{dem}_1 \rightarrow \mathit{demanswer}_1\ ;;$
...
$\mathit{dem}_m \rightarrow \mathit{demanswer}_m\ ;;$

Note that the answers for a demand is only printed *iff* the demand is valid for that instant.

If the `--uuid` option is set, then the UUID of the demands is printed out, as it is for equations:

```
// Equations
eqnuuid_1 ;;
...
eqnuuid_l ;;

// Expressions
answer_1 ;;
...
answer_n ;;

// Demands
dem_1 → demuuid_1 → demanswer_1 ;;
...
dem_m → demuuid_m → demanswer_m ;;
```

Should the `--verbose` option also be set, then the results will be of the form

```
// Equations
eqn_1 → eqnuuid_1 ;;
...
eqn_ℓ → eqnuuid_ℓ ;;

// Expressions
expr_1 → answer_1 ;;
...
expr_n → answer_n ;;

// Demands
dem_1 → demuuid_1 → demanswer_1 ;;
...
dem_m → demuuid_m → demanswer_m ;;
```

6.2 Time in `tlcore`

Formalizing the behavior of the `tlcore` is quite simple. In `tlcore`, at each instant n, there is a current set of equations σ_n, and then a number of expressions E_{nk} that need to be evaluated. We can therefore consider that at each instant, for each expression, we have a simplified version of a Core TransLucid program, with no local `where` clauses.

To be able to define these σ_n, we will build *sets of identified equations* Ω, where $\Omega \in \mathbf{U} \times \mathbb{N} \times \mathbb{N} \times \mathbf{Q}$, where $\mathbf{U}$ ($\ni u$) is the set of possible UUIDs and $\mathbf{Q}$ ($\ni q$) is the set of possible equations. The elements of Ω are of the form $r = (u_r, i_r, j_r, q_r)$, meaning that for UUID u_r, for all t such that $i_r \leqslant t \leqslant j_r$, the current equation is q_r.

Given a current set of identified equations Ω, one can create a new set Ω' through one of the following operations: `add`(q), `del`(u) or `repl`(u, q). Suppose the current instant is n. Then:

- `add`(q): Pick a $u \in \mathbf{U}$ such that $u \notin \mathrm{dom}(\Omega)$.

$$\Omega' = \Omega \cup \big\{(u, n, \infty, q)\big\}$$

- `del`(u):

$$\Omega' = \{r \mid r \in \Omega \wedge u_r \neq u\} \cup \big\{(u_r, i_r, j, q_r) \mid r \in \Omega \wedge j = \min(j_r, n-1)\big\}$$

- `repl`(u, q):

$$\Omega' = \{r \mid r \in \Omega \wedge u_r \neq u\} \cup \big\{(u_r, i_r, j, q_r) \mid r \in \Omega \wedge j = \min(j_r, n-1)\big\} \cup \big\{(u, n, \infty, q)\big\}$$

A set of these changes is written ς. We define two sequences $\Omega_0, \Omega_1, \ldots$ and $\Omega'_0, \Omega'_1, \ldots$ of identified equations, where:

- $\Omega_0 = \varnothing$;
- $\Omega_{n+1} = \Omega'_n$;
- $\Omega'_n = \varsigma_n \Omega_n$;
- $\sigma_n = \big\{q \mid (u, i, j, q) \in \Omega'_n \wedge i \leqslant t \leqslant j\big\}$.

Now, for each instant n, expression E_{nk} is evaluated with respect to the set of equations σ_n.

6.3 The S^3 application

The S^3 application provides the means for exploring a multidimensional hyperdaton defined using the TransLucid runtime system. The idea is that there is a collection of equations Q and an expression E using the variables defined in Q, and that the user should be able to explore the multidimensional space and see what the values of E are as the user changes the settings of the multiple definitions in play.

6.3.1 Overview

In addition to defining a current header H, a current set of equations Q and a current expression E, the user must define a relevant set of dimensions $d_1, \ldots, d_n$ (some of these may be defined in the header). To simplify the presentation, it will be assumed that all dimensions are identifiers, that the values of all dimensions are integer-valued, and that the user is always viewing a contiguous range of these integer-valued dimensions.

Visualization of the results of E is done through a two-dimensional table. Given that n is most likely greater than two, two of the n dimensions must be designated to vary, while the others will be fixed. The manner in which this is done is to define a current *pivot point* p, about which exploration takes place. This pivot point is in fact simply an n-dimensional tuple, fixing the values for all dimensions.

Let d_h be the horizontal dimension and d_v the vertical dimension. The user must define how much variance around $p(d_h)$ and around $p(d_v)$ must take place, by defining the *horizontal radius* s_h and the *vertical radius* s_v. Then the values of dimension d_h are allowed to vary between $p(d_h) - s_h$ and $p(d_h) + s_h$, while those of dimension d_v are allowed to vary between $p(d_v) - s_v$ and $p(d_v) + s_v$.

6.3.2 Example

We now consider an example.

1. Suppose the user has provided the following information:

Dimension	Vertical(2)	Horizontal(4)	Pivot
x	•(2–6)		4
y		•(3–11)	7
w			11
z			42

Here x is defined to be the vertical dimension, with pivot value 4 and vertical radius 2, while y is defined to be the horizontal dimension, with pivot value 7 and horizontal radius 4. Dimensions w and z are respectively with pivot values 11 and 42.

Suppose that Q is empty and that E is $\texttt{\#}w + \texttt{\#}z + \texttt{\#}y - \texttt{\#}x$. Then the table will look this:

$x\backslash y$	3	4	5	6	7	8	9	10	11
2	54	55	56	57	58	59	60	61	62
3	53	54	55	56	57	58	59	60	61
4	52	53	54	55	56	57	58	59	60
5	51	52	53	54	55	56	57	58	59
6	50	51	52	53	54	55	56	57	58

The top left-hand corner corresponds to

$$(\texttt{\#}w + \texttt{\#}z + \texttt{\#}y - \texttt{\#}x)\ \texttt{@}\ [w:11, z:42, y:3, x:2] = 54$$

while the bottom right-hand corner corresponds to

$$(\texttt{\#}w + \texttt{\#}z + \texttt{\#}y - \texttt{\#}x)\ \texttt{@}\ [w:11, z:42, y:11, x:6] = 58$$

2. Suppose now the user changes the values for w and z:

Dimension	Vertical(2)	Horizontal(4)	Pivot
x	•(2–6)		4
y		•(3–11)	7
w			12
z			38

The recalculation gives:

$x\backslash y$	3	4	5	6	7	8	9	10	11
2	51	52	53	54	55	56	57	58	59
3	50	51	52	53	54	55	56	57	58
4	49	50	51	52	53	54	55	56	57
5	48	49	50	51	52	53	54	55	56
6	47	48	49	50	51	52	53	54	55

3. Suppose now the user makes w the horizontal dimension:

Dimension	Vertical(2)	Horizontal(4)	Pivot
x	•(2–6)		4
y			7
w		•(8–16)	12
z			38

The recalculation gives:

$x\backslash w$	8	9	10	11	12	13	14	15	16
2	51	52	53	54	55	56	57	58	59
3	50	51	52	53	54	55	56	57	58
4	49	50	51	52	53	54	55	56	57
5	48	49	50	51	52	53	54	55	56
6	47	48	49	50	51	52	53	54	55

4. Suppose now the user makes z the vertical dimension:

Dimension	Vertical(2)	Horizontal(4)	Pivot
x			4
y			7
w		•(8–16)	12
z	•(36–40)		38

The recalculation gives:

$z\backslash w$	8	9	10	11	12	13	14	15	16
36	47	48	49	50	51	52	53	54	55
37	48	49	50	51	52	53	54	55	56
38	49	50	51	52	53	54	55	56	57
39	50	51	52	53	54	55	56	57	58
40	51	52	53	54	55	56	57	58	59

5. Suppose now the user shrinks the horizontal spread to 3:

Dimension	Vertical(2)	Horizontal(3)	Pivot
x			4
y			7
w		•(9–15)	12
z	•(36–40)		38

The recalculation gives:

$z\backslash w$	9	10	11	12	13	14	15
36	48	49	50	51	51	52	53
37	49	50	51	52	53	54	55
38	50	51	52	53	54	55	56
39	51	52	53	54	55	56	57
40	52	53	54	55	56	57	58

6. Finally, the user changes E to $\#z * \#x - \#w * \#y$.

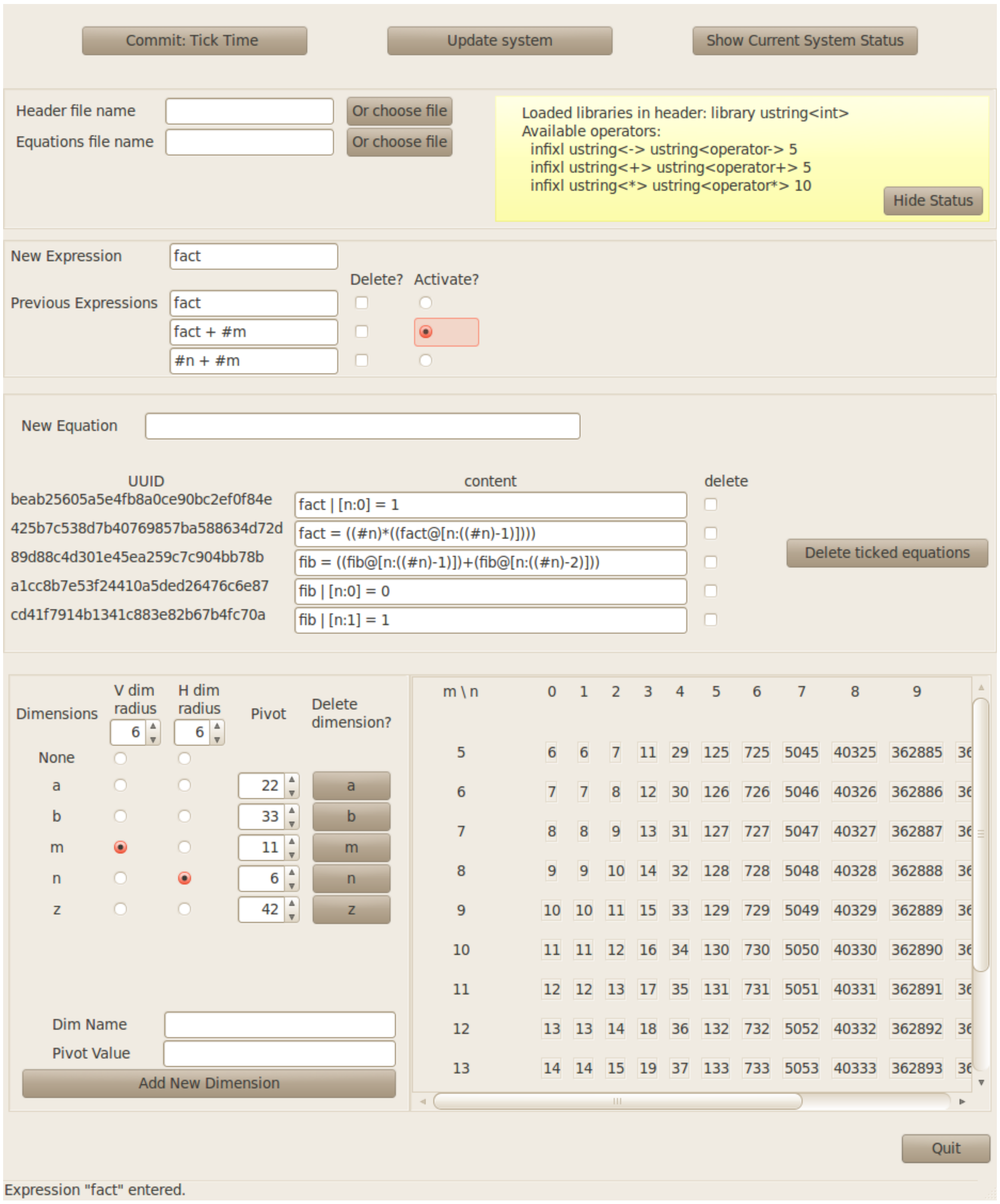
$z\backslash w$	9	10	11	12	13	14	15
36	81	74	67	60	53	46	39
37	85	78	71	64	57	50	43
38	89	82	75	68	61	54	47
39	93	86	79	72	65	58	51
40	97	90	83	76	69	62	55

6.3.3 Code browsing

The $\mathtt{S}^3$ application does much more than exploring single hyperdatons. It is in fact a full-fledged TransLucid code browser, allowing for the interactive editing of systems of TransLucid equations, with complete control through the use of UUIDs.

Figure 6.1 gives an example of the use of $\mathtt{S}^3$. It shows the current state of the system after a header file has been loaded, along with two files containing equations, one for factorial, one for Fibonacci. The UUIDs for the equations and the printout of the equations have been provided by the TransLucid runtime library, as have the evaluations provided in the table in the lower right-hand side.

The system is ready to show three different expressions, *fact*, *fact* + $\#m$, and $\#n + \#m$. The pivot is set to $[a : 22, b : 33, m : 11, n : 6, z : 42]$, with vertical and horizontal radii of 6.



Header file name

Equations file name

Loaded libraries in header: library ustring<int>
Available operators:
  infixl ustring<-> ustring<operator-> 5
  infixl ustring<+> ustring<operator+> 5
  infixl ustring<*> ustring<operator*> 10

New Expression	fact		
		Delete?	Activate?
Previous Expressions	fact	☐	○
	fact + #m	☐	●
	#n + #m	☐	○

New Equation

UUID	content	delete
beab25605a5e4fb8a0ce90bc2ef0f84e	fact | [n:0] = 1	☐
425b7c538d7b40769857ba588634d72d	fact = ((#n)*((fact@[n:((#n)-1)])))	☐
89d88c4d301e45ea259c7c904bb78b	fib = ((fib@[n:((#n)-1)])+(fib@[n:((#n)-2)]))	☐
a1cc8b7e53f24410a5ded26476c6e87	fib | [n:0] = 0	☐
cd41f7914b1341c883e82b67b4fc70a	fib | [n:1] = 1	☐

Dimensions	V dim radius	H dim radius	Pivot	Delete dimension?
	6	6		
None	○	○		
a	○	○	22	a
b	○	○	33	b
m	●	○	11	m
n	○	●	6	n
z	○	○	42	z

Dim Name

Pivot Value

m \ n	0	1	2	3	4	5	6	7	8	9	
5	6	6	7	11	29	125	725	5045	40325	362885	36
6	7	7	8	12	30	126	726	5046	40326	362886	36
7	8	8	9	13	31	127	727	5047	40327	362887	36
8	9	9	10	14	32	128	728	5048	40328	362888	36
9	10	10	11	15	33	129	729	5049	40329	362889	36
10	11	11	12	16	34	130	730	5050	40330	362890	36
11	12	12	13	17	35	131	731	5051	40331	362891	36
12	13	13	14	18	36	132	732	5052	40332	362892	36
13	14	14	15	19	37	133	733	5053	40333	362893	36

Expression "fact" entered.

Figure 6.1: Example of use of S^3.

The browser is designed to allow the user to experiment with the whole system. Therefore, the approach to time taken in `tlcore` is too rigid. In S^3, the instants used by `tlcore` are only advanced once the user has decided that the set of equations has reached a sufficiently stable condition, i.e., once the user has pressed the `Commit: Tick Time` button. This button commits the changes and advances time.

6.3.4 Evolution

The S^3 tool is very experimental, and many additions are being considered, to cover the code browsing experience, but more importantly, the multidimensional exploration. We envisage that the tables of values will only be one form of output. Graphs, maps and other visual presentations are possible, as is navigation with dimensions that are not integer-valued. In time, multiple S^3 tools will be able to interact, collectively programming large TransLucid projects.

6.4 Time in S^3

To support applications like S^3, the TransLucid runtime library cannot behave in the same way as for `tlcore`. In `tlcore`, each instant is programmed in batch mode, then executed. In S^3, each (macro-)instant consists of a number of micro-instants, but once the `Commit` button has been pressed, then the macro-instant is finalized, as in `tlcore`.

The idea is that the changes made since the beginning of the current instant are all tentative, and are only finalized once the `Commit` button has been pressed. Any changes that are no longer seen at the end of that instant are no longer accessible.

When the system is fully built, S^3 will allow navigation through previous instants as well, with `Reset instant` and `Goto previous instant` buttons.

Formally, within each macro-instant n, there is a sequence of changes ς_{nm}, $1 \leqslant m \leqslant k_n$. We will write $\varsigma_{n\infty}$ for ς_{nk_n}. Then we define

- $\Omega_0 = \varnothing$;
- $\Omega_{n+1} = \Omega'_n$;
- $\Omega'_n = \Omega_{n\infty}$;
- $\Omega_{n0} = \Omega_n$;
- $\Omega_{nm} = \varsigma_{nm}\ \Omega_{n(m-1)}$;
- $\sigma_n = \big\{q \mid (u,i,j,q) \in \Omega'_n \wedge i \leqslant t \leqslant j\big\}$.

Hence, once the macro-instant n is finalized, expression E_{nk} is evaluated with respect to the set of equations σ_n. Furthermore, all of the changes that took place within the macro-instant become invisible to future instants.

This is not the first time that we see micro-instants. The advance of time in the cached operational semantics (§4.6) also corresponds to micro-instants. This means that the TransLucid runtime system can advance micro-instants either through its own behavior or because of S^3 telling it to do so. To maintain determinism, therefore, TransLucid cannot provide any information about the advance of its micro-instants to callers of the library.

The cached semantics uses multiple clocked threads, each of which advances automatically if it discovers that another thread or the cache has a clock that is ahead of it. These two notions of micro-instant can be combined, should no outside entity be aware of the internal advance of micro-instants. It suffices to ensure that the updates to the set of equations are fully interleaved with the computations. In other words, either an equation update is being made, or computation is taking place. In the second case, then the equation update is simply assumed to be the empty set.

6.5 Adding context

Chapter 2 presented the idea of the æther. Inspired by the luminiferous æther that was the basis for nineteenth century research in electromagnetism, the idea of an æther for computation was that it creates an immersive context in which all calculations 'bathe', like the water flowing through biological cells. The question is, what is the relevance to TransLucid?

Up to now, all of the writing about TransLucid, whether informal presentation, formal semantics, or informal presentation of the implementation, has supposed that the evaluation of expressions is context-dependent. But is it possible to make an entire *system* context-dependent? The answer is yes, by introducing an *inner æther*, special variable `ÆTHER`, which is programmed as a variable, but accessed like a dimension.

The idea is that at time t, `#ÆTHER` will simply evaluate the variable `ÆTHER` with no context information apart from the current value of `time`. Since, `ÆTHER` is a variable, it can have subvariables, and these can have subvariables, so that this runtime context can have as much substructure as is necessary.

During instant t, during the evaluation of expressions, some new equations for `ÆTHER` can be generated. These do not contribute to the bestfitting for `ÆTHER` at macro-instant t; they only become relevant at macro-instant $t + 1$.

Using this approach, the æther can be programmed jointly by all components of a system as well as from the outside. The bestfitting inherent in the way that TransLucid works ensures that if multiple definitions are made, there will be some meaningful outcome.

6.6 Systems within systems

Furthermore, if we look at the current implementation, we can see that class `SystemHD` is a subclass of class `VariableHD`, and `VariableHD` can hold several other variables below it. What this implies is that a system can contain other systems. But, given that these systems run with a synchronous timed semantics, how would this work and how would it be compatible with the idea of instants and micro-instants of the above discussion?

The solution lies in providing a system with a *clock level* $\ell \in \mathbb{Z}$. Clocks of level ℓ advance with ticks of size ϵ^ℓ, where ϵ is an infinitesimal level. All clocks of the same clock level would be understood as running "fairly close" to each other, in other words, at most finitely faster or slower one from each other. However, a system B contained within a system A would run with a clock level of one plus the clock level of A. If the clock level of A is ℓ, that of B would be $\ell + 1$, i.e., B would run infinitely faster than A. C contained within B would run at level $\ell + 2$, and so on.

Each system has its own physical inputs and outputs, defined externally. As a result, we now have a model where multiple levels of system can be defined, with the inner levels running with faster clock rates. The relation between macro-instants and micro-instants is simple: a system running at clock level ℓ has level-ℓ macro-instants and level-$(\ell + 1)$ micro-instants.

Choosing *infinitely* faster clocks has the same simplicity as does the original synchronous model. Were one to choose a finite amount, the immediate next question would be, "How much?", and the answer would change from one physical implementation to the next. By stating that the clock is infinitely faster, we need not worry about such problems.

However, in a given level-ℓ macro-instant, we cannot allow an infinite number of level-$(\ell + 1)$ micro-instants to take place, as this would mean an infinite amount of work would have to take place in a finite amount of physical time; this clearly is not possible. Therefore, some upper bound would have to be placed on the number of micro-instants for each level. Demands not completed within this number of micro-instants would, depending on the requests, either timeout or else be continued in subsequent macro-instants.

6.7 Time in multiple TransLucid systems

The fact that we can understand the timing of systems embedded at different levels does not in any way mean that we know how to program them. We have come up with three different models for multiple TransLucid systems interacting. We call them the *peer-to-peer* model, the *hierarchical-worker* model and the *parent-æther* model.

In the peer-to-peer model, two or more systems, at the same clock level, interact by writing equations and demands for each other. Each system reacts perfectly synchronously, so will only see the "inputs" from the other systems at the beginning of an instant. Therefore, there can be no instantaneous chatter between two systems. It works by the following principles:

- If p_1 and p_2 are two peers, then the clock level of p_1 is the same as the clock level of p_2.
- Each instant for p_1 (or p_2) is synchronous in behavior, i.e., while processing an instant, the set of equations may not change.
- The real time between the end of instant i and the beginning of instant $i + 1$ of p_1 (or p_2) is non-null. This is a fairnesss assumption.
- There is a maximum real delay between the request for a tick of p_1 (or p_2) before an instant actually begins. This is a lack-of-starvation assumption.

In the hierarchical-worker model, two or more systems lie within another system, and the inner systems respond to equations and demands provided to them by the outer system. Since the micro-instants of the outer system are on the same level as the macro-instants of the inner systems, this communication is possible. It works by the following principles:

- If p_1 contains p_2, then the clock level of p_2 is one more than the clock level of p_1.
- The inner system p_2 will be invoked by a thread w of p_1. For w, executing p_2 is instantaneous, i.e., if the invocation of p_2 takes place at time t of w, the result from p_2 will be provided at time t as well.
- If instant i of p_2 takes place at time t, instant $i + 1$ takes place at the earliest at time $t + 1$.

In the parent-æther model, two or more systems lie within another system, and the inner systems program the outer system. For this to work, the inner systems must "agree" on the equations and demands provided to the outer system. In this sense, the actions of the inner systems are tentative, and resemble the modifications made by the user of the $\mathtt{S}^3$ system. In this situation, the "commit" can only take place when all of the inner systems concur on the changes to be made to the parent system. An example of such a system would be a central code base for multiple $\mathtt{S}^3$ systems. It works by the following principles:

- If p_1 contains p_2 and p_3, then the clock level of p_2 and p_3 is one more than the clock level of p_1.
- If p_1 contains p_2 and p_3, then p_2 and p_3 act as peers with respect to each other.
- The changes to p_1 made by p_2 and p_3 take place through an equation registry, whose timed behavior is as for the cache in the cached operational semantics.
- The macro-instants of p_1 advance when one or both of p_2 or p_3 commits its changes and there are no conflicts.

It must be made clear that the ideas developed up to now about multiple timed systems could only have come about through the development of physical applications. We expect these to become clearer as we polish the current applications and develop new ones. The same is even more true for developing timed primitives for TransLucid in order to manipulate the different kinds of distributed synchronous systems described above.

Chapter 7

Bottom-up Recursion

In this chapter, we focus on efficient implementations of recursively defined data structures or functions, by their transformation into bottom-up iterative structures. We consider simple recursive functions, recursively defined dataflow filters, and divide-and-conquer and dynamic programming algorithms.

7.1 Simple recursive functions

We begin by considering the three standard recursive functions presented in Chapter 3.

7.1.1 Factorial

Consider the factorial function from Chapter 3:

$$\begin{array}{l} \mathit{fact}.n = f \\ \texttt{where} \\ \quad f = 1\ \texttt{fby}.n\ f * \texttt{index}.n\ ;; \\ \texttt{end} \end{array}$$

The recursive implementation takes $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ stack. If we memoize when calculating, then calculation still takes $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ stack, and $\mathcal{O}(n)$ cache.

If we wish to generate efficient code from this function, we need to replace the recursive, stack-based, top-down evaluation by an iterative, stackless, bottom-up evaluation. We can do this by using one counter to keep track of n and one memory to keep track of the previous element implicit in the `fby`.n.

```
template <Dimension d>
unsigned
fact (Context kappa)
{
  unsigned mem0 = 1;
  unsigned d_count = 0;
  unsigned d_ord = kappa(d);
  while (d_count != d_ord)
  {
    ++d_count;
    mem0 = d_count * mem0;
  }
  return mem0;
}
```

If we are going to call this function a lot, for example to produce a table, then we can cache the memory with the assumption that later calls will be for larger values of n.

```
template <Dimension d>
unsigned
fact (Context kappa)
{
  static unsigned mem0 = 1;
  static unsigned d_count = 0;
  unsigned d_ord = kappa(d);
  if (d_count > d_ord)
  {
    mem0 = 1;
    d_count = 0;
  }
  while (d_count != d_ord)
  {
    ++d_count;
    mem0 = d_count * mem0;
  }
  return mem0;
}
```

This solution now gives sublinear amortized performance time and runs in constant space.

7.1.2 Fibonacci

Consider the Fibonacci function from Chapter 3:

$$\begin{array}{l} \mathit{fib}.n = f \\ \texttt{where} \\ \quad f = 0\ \texttt{fby}.n\ 1\ \texttt{fby}.n\ f + \texttt{next}.n\ f\ ;; \\ \texttt{end} \end{array}$$

The recursive implementation takes $\mathcal{O}(\mathit{fib}(n))$ time and $\mathcal{O}(n)$ stack. If we memoize when calculating, then calculation takes $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ stack, and $\mathcal{O}(n)$ cache.

The bottom-up solution matching that of factorial requires one counter (n), two memories implicit in the $\texttt{fby}.n$ and a temporary variable for $f + \texttt{next}.n\ f$.

```
template <Dimension d>
unsigned
fib (Context kappa)
{
  unsigned mem0 = 0;
  unsigned mem1 = 1;
  unsigned d_count = 1;
  unsigned d_ord = kappa(d);
  if (d_ord==0)
    return mem0;
  while (d_count != d_ord)
  {
    ++d_count;
    unsigned temp = mem0 + mem1;
    mem0 = mem1;
    mem1 = temp;
  }
  return mem1;
}
```

As with factorial, we can cache previously computed results. If successive calls to Fibonacci are with larger inputs, amortized performance is sublinear.

```
  template <Dimension d>
  unsigned
  fib (Context kappa)
  {
    static unsigned mem0 = 0;
    static unsigned mem1 = 1;
    static unsigned d_count = 1;
    unsigned d_ord = kappa(d);
    if (d_count > d_ord)
    {
      unsigned mem0 = 0;
      unsigned mem1 = 1;
      unsigned d_count = 1;
    }
    if (d_ord==0)
      return mem0;
    while (d_count != d_ord)
    {
      ++d_count;
      unsigned temp = mem0 + mem1;
      mem0 = mem1;
      mem1 = temp;
    }
    return mem1;
  }
```

7.1.3 Ackermann

Consider the Ackermann function from Chapter 3:

$$\begin{array}{l} ack.m.n = f \\ \texttt{where} \\ \quad f = \texttt{index}.n\ \texttt{fby}.m\ (\texttt{next}.n\ f\ \texttt{fby}.n\ \texttt{next}.m\ f)\ ;; \\ \texttt{end} \end{array}$$

The Ackermann function is one of the simplest recursive functions that is not primitive recursive, i.e., cannot be transformed into a single `while` loop. The default implementation takes $\mathcal{O}(ack(m,n))$ time, $\mathcal{O}(ack(m,n))$ cache.

Notice that for every m, there is a need for a counter (n) and a memory ($\texttt{fby}.n$). Hence, to compute $\texttt{ack}_m$, we need m counters and m memories. Here is a hand-coded solution for $\texttt{ack}_3$:

```
ack3 (unsigned n)
{
  register unsigned i1, i2, i3;
  register unsigned a1 = 1, a2, a3;

  // ack 2 0 = ack 1 1 = 3
  for (i1 = 0;  i1 <= 1; ++i1,++a1);
  a2 = a1;

  // ack 3 0 = ack 2 1 = 5
  for (; i1 <= a2; ++i1,++a1);
  a3 = a2 = a1;
  i2 = 2;

  // ack 3 1 ... ack 3 n
  for (i3 = 1;  i3 <= n; ++i3)
  {
    for (; i2 <= a3; ++i2)
    {
      for (; i1 <= a2; ++i1,++a1);
      a2 = a1;
    }
    a3 = a2;
  }
  return a3;
}
```

There are two problems with this solution. The first is that it is very hard to write and debug. The second is that it is not a general solution: we need to write such a program for each m. However, it is *blazingly* fast. The question is, "Is it possible to generate a solution that has similar efficiency?" The answer is yes, if we use `C++` templates. The m argument is passed as a template parameter, while the n argument is passed as an argument to the `operator()` method. This solution relies on the memoize-and-reset approach introduced with the factorial and Fibonacci functions.

```
template <Dimension m, n, unsigned m_ord>
class ack
{
  private:
    unsigned n_count;
    unsigned mem;
    ack<m,n,m_ord-1> ack_mminus;

  public:
    ack<m,n,m_ord>()
    {
      reset();
    }

    void
    reset()
    {
      n_count = 0;
      mem = ack_mminus(kappa.delta(n,1));
    }
    unsigned
    operator()(Context kappa)
    {
      unsigned n_ord = kappa(n);
      if (n_count > n_ord)
        reset();
      return calc(n_ord);
    }

    unsigned
    calc(unsigned n_ord)
    {
      while (n_count != n_ord)
      {
        n_count++;
        mem = ack_mminus.calc(mem);
      }
      return mem;
    }
};

template<>
class ack<m,n,0>
{
  public:

    ack<0>()
    {}

    unsigned
    operator()(Context kappa)
    {
      unsigned n_ord = kappa(n);
      return calc(n_ord);
    }

    unsigned
    calc(unsigned n_ord)
    {
      return n_ord+1;
    }
};
```

This solution is slower than the hand-coded `C` program, but it is of the same order of magnitude (9% slower on `ack 3 29`). If one attempts to run this function on most languages and compilers, one gets perhaps to `ack 3 13`!

This solution still suffers from the fact that one needs to know what the value of n is at compile time. If this is not known, then another version, using a dynamic n and a pointer to the `ack_mminus` can be written. The continual indirection through the pointer, and the fact that the compiler cannot optimize completely, slow down the result substantially; nevertheless, it is still a lot faster and compact than a recursive implementation.

Most importantly, we have moved down to constant-size memory and cache, and sublinear amortized time in the second argument.

7.1.4 The general case

We are now ready to consider the general case.

```
$general.d = f$
where
  $f = G_0(0)$ fby.$d$ ... fby.$d$ $G_{k-1}(k-1, f_0, \ldots, f_{k-2})$ fby.$d$ $G_k\big(\#d, f_{[d:\#d-k]}, \ldots, f_{[d:\#d-1]}\big)$
end
```

where $G_0, \ldots, G_k$ are strict functions. The translation is as follows:

```
template <Dimension d>
class general
{
  private:
    unsigned d_count;
    T mem_0, ..., mem_{k-1};
  public:
    general<d>()
    {
      reset();
    }
    void
    reset()
    {
      T mem_0 = G_0(0);
      ...
      T mem_{k-1} = G_{k-1}(k-1,mem_0,...,mem_{k-2});
      unsigned d_count = k-1;
    }
    T operator() (Context kappa)
    {
      unsigned d_ord = kappa(d);
      if (d_count > d_ord && d_count >= k)
        reset();
      switch (d_ord)
      {
        case 0:  return mem_0;
        ...
        case k-1:  return mem_{k-1};
        default:
      }
      while (d_count != d_ord)
      {
        ++d_count;
        unsigned temp = G_k(d_count,mem_0,...,mem_{k-1});
        mem_0 = mem_1;
        ...
        mem_{k-1} = temp;
      }
      return mem_{k-1};
    }
};
```

7.2 Dataflow filters

Consider the dataflow filters from Chapter 3:

```
$X$ wvr.$d$ $Y$ =
  if first.$d$ $Y$
  then $X$ fby.$d$ (next.$d$ $X$ wvr.$d$ next.$d$ $Y$)
  else next.$d$ $X$ wvr.$d$ next.$d$ $Y$ fi ;;

$X$ upon.$d$ $Y$ =
  $X$ fby.$d$ if first.$d$ $Y$
          then next.$d$ $X$ upon.$d$ next.$d$ $Y$
          else $X$ upon.$d$ next.$d$ $Y$ fi ;;

$X$ $merge$.$d$ $Y$ =
  if first.$d$ $X$ < first.$d$ $Y$
    then $X$ fby.$d$ (next.$d$ $X$ $merge$.$d$ $Y$)
  elsif first.$d$ $X$ > first.$d$ $Y$
    then $Y$ fby.$d$ ($X$ $merge$.$d$ next.$d$ $Y$)
  else $Y$ fby.$d$ (next.$d$ $X$ $merge$.$d$ next.$d$ $Y$) fi ;;
```

7.2.1 Indexical definitions

In the papers presenting Indexical Lucid [37, 5], it is shown that these filters can be converted into a set of recursive Lucid flows, without any recursive instantiation of the filters:

```
$X$ wvr.$d$ $Y$ = $X$ @ $[d : T]$
  where
    $T$ = $U$ fby.$d$ $U$ @ $[d : T + 1]$ ;;
    $U$ = if $Y$ then #$d$ else next.$d$ $U$ fi ;;
  end

$X$ upon.$d$ $Y$ = $X$ @ $[d : W]$
  where
    $W$ = 0 fby.$d$ if $Y$ then $W + 1$ else $W$ fi ;;
  end

$X$ $merge$.$d$ $Y$ =
if $xx \leqslant yy$ then $xx$ else $yy$ fi
  where
    $xx$ = $x$ upon.$d$ $xx \leqslant yy$ ;;
    $yy$ = $y$ upon.$d$ $yy \leqslant xx$ ;;
  end
```

7.2.2 Iterative implementations

We apply the memoize-with-reset technique to the dataflow filters, using templated classes: the dimension is a template argument and the two hyperdatons are constructor arguments.

In order to simplify the presentation, we suppose that X and Y are infinite. In a production system, they would need to be modified to take into account finite streams, but so would the original recursive definitions. We also do not consider situations where part of the past might have been erased, thereby making resetting impossible.

```
template <Dimension d>
class wvr
{
  private:
    HD* X;  HD* Y;
    unsigned d_count;
    unsigned d_rank;

  public:
    wvr(HD* X_in, HD* Y_in)
    : X(X_in), Y(Y_in), d_count(0)
    { }

    void reset()
    {
      d_rank = 0;
      while ((*Y)(kappa.delta(d,d_rank)) == false)
        ++d_rank;
    }

    TypedValue operator()(Context kappa)
    {
      unsigned d_ord = kappa(d);
      if (d_count > d_ord)
        d_count = 0;
      if (d_count == 0)
        reset();
      while (d_count != d_ord)
      {
        ++d_count;
        while ((*Y)(kappa.delta(d,d_rank)) == false)
          ++d_rank;
      }
      return (*X)(kappa.delta(d,d_rank));
    }
};

template <Dimension d>
class upon
{
  private:
    HD* X;  HD* Y;
    unsigned d_count;
    unsigned d_rank;

  public:
    upon(HD* X_in, HD* Y_in)
    : X(X_in), Y(Y_in), d_count(0)
    { }

    void reset()
    {
      d_rank = 0;
    }
    TypedValue operator()(Context kappa)
    {
      unsigned d_ord = kappa(d);
      if (d_count > d_ord)
        d_count = 0;
      if (d_count == 0)
        reset();
      while (d_count != d_ord)
      {
        ++d_count;
        if ((*Y)(kappa.delta(d,d_rank)) == true)
          ++d_rank;
      }
      return (*X)(kappa.delta(d,d_rank));
    }
};

template <Dimension d>
class merge
{
  private:
    HD* X;  HD* Y;
    unsigned d_count;
    unsigned d_rankX, d_rankY;

  public:
    merge(HD* X_in, HD* Y_in)
    : X(X_in), Y(Y_in), d_count(0)
    { }

    TypedValue operator()(Context kappa)
    {
      unsigned d_ord = kappa(d);
      if (d_count > d_ord)
        d_count = 0;
      if (d_count == 0)
        reset();
      while (d_count != d_ord)
      {
        ++d_count;
        if ((*X)(kappa.delta(d,d_rankX)) < (*Y)(kappa.delta(d,d_rankY)))
          ++d_rankX;
        else if((*X)(kappa.delta(d,d_rankX)) > (*Y)(kappa.delta(d,d_rankY)))
          ++d_rankY;
        else
        {
          ++d_rankX;
          ++d_rankY;
        }
      }
      return min((*X)(kappa.delta(d,d_rankX)),(*Y)(kappa.delta(d,d_rankY)));
    }
    void reset()
    {
      d_rankX = 0;
      d_rankY = 0;
    }
};
```

7.2.3 The general case

Here is the general case for dataflow filters:

```
general.d $(X_1, \ldots, X_n)$ =
  $G_1(X_1, \ldots, X_n)$ fby.d $\cdots$ fby.d $G_k(X_1, \ldots, X_n)$
  fby.d if $cond_0$(first.d $X_1, \ldots,$ first.d $X_n$)
          then $G_{0,1}(X_1, \ldots, X_n)$ fby.d $\cdots$ fby.d $G_{0,k_0}(X_1, \ldots, X_n)$
                fby.d general.d (next$^{a_{0,1}}$ $X_1, \ldots,$ next$^{a_{0,n}}$ $X_n$)
        ...
        elsif $cond_{m-1}$(first.d $X_1, \ldots,$ first.d $X_n$)
          then $G_{m-1,1}(X_1, \ldots, X_n)$ fby.d $\cdots$ fby.d $G_{m-1,k_{m-1}}(X_1, \ldots, X_n)$
                fby.d general.d (next$^{a_{m-1,1}}$ $X_1, \ldots,$ next$^{a_{m-1,n}}$ $X_n$)
          else $G_{m,1}(X_1, \ldots, X_n)$ fby.d $\cdots$ fby.d $G_{m,k_m}(X_1, \ldots, X_n)$
                fby.d general.d (next$^{a_{m,1}}$ $X_1, \ldots,$ next$^{a_{m,n}}$ $X_n$)
```

and here is the translation:

```
template <Dimension d>
class general
{
  private:
    HD* X1; ...; HD* Xn;
    unsigned d_count;
    unsigned d_rankX1, ..., d_rankXn;

  public:
    general(HD* X1_in, ..., HD* Xn_in)
    :  X1(X1_in), ..., Xn(Xn_in), d_count(0)
    { }

    void reset()
    {
      d_rankX1 = 0;
      ...
      d_rankXn = 0;
    }
  operator()(Context kappa)
  {
    unsigned d_ord = kappa(d);
    if (d_count > d_ord)
      d_count = 0;
    if (d_count == 0)
      reset();
    while (d_count <= d_ord)
    {
      if (d_count+$k$ >= d_ord)
        return $G_{\texttt{d\_ord-d\_count}}$((*X$_i$)(kappa.delta(d,d_rank X$_i$)))
      if ($cond_0$((*X$_i$)(kappa.delta(d,d_rank X$_i$))))
      {
        if (d_count+$k$+$k_0$ >= d_ord)
          return $G_{0,\texttt{d\_ord-d\_count-}k}$((*X$_i$)(kappa.delta(d,d_rank X$_i$)))
        else
        {
          d_rankX1 += $a_{0,1}$;
          ...
          d_rankXn += $a_{0,n}$;
          d_count += $k$+$k_0$;
        }
      }
      ...
      else if ($cond_{m-1}$((*X$_i$)(kappa.delta(d,d_rank X$_i$))))
      {
        if (d_count+$k$+$k_{m-1}$ >= d_ord)
          return $G_{m-1,\texttt{d\_ord-d\_count-}k}$((*X$_i$)(kappa.delta(d,d_rank X$_i$)))
        else
        {
          d_rankX1 += $a_{m-1,1}$;
          ...
          d_rankXn += $a_{m-1,n}$;
          d_count += $k$+$k_{m-1}$;
        }
      }
      else
      {
        if (d_count+$k$+$k_m$ >= d_ord)
          return $G_{m,\texttt{d\_ord-d\_count-}k}$((*X$_i$)(kappa.delta(d,d_rank X$_i$)))
        else
        {
          d_rankX1 += $a_{m,1}$;
          ...
          d_rankXn += $a_{m,n}$;
          d_count += $k$+$k_m$;
        }
      }
    }
    // Should never get here.
  }
};
```

7.3 Divide-and-conquer algorithms

The previous sections showed that for recursive functions with a linear structure, these can be efficiently implemented by not following a naïve solution. In this section, we look at divide-and-conquer algorithms,[1] where a problem is broken up into subproblems of size significantly smaller than the original problem, typically with the same algorithm then being called on the smaller pieces. The recursion is then branching, and the arguments can be transformed into variance in space or time dimensions, thereby allowing bottom-up computations, often in parallel. We do not go into all of the details.

7.3.1 Factorial, version 2

Factorial of n can be written

$$\prod_{i=0}^{n} i.$$

To compute this number, we can divide-and-conquer, knowing that:

$$\prod_{i=m}^{n} i = \prod_{i=m}^{n'} i \; * \prod_{i=n'+1}^{n} i,$$

where $n' = \lfloor \frac{m+n}{2} \rfloor$.

```
$fact2.d =$
   if $(\#d) \equiv 0$ then $1$ else $f.d.1.(\#d)$
   where
      $f.d.m.n =$
         if $m \equiv n$ then $m$ else $f.d.m.n' * f.d.(n'+1).n$
         where
            $n' = (m+n)/2$ ;;
         end
   end
```

This program can be transformed into a bottom-up solution, using a temporary dimension t. The solutions are not identical, as the solution below branches on powers of 2.

```
$fact3.d =$
   if $(\#d) \equiv 0$ then $1$ else $f.d.(\#d)$
   where
      $f.d.n = B$ @ $[d:0, t:k]$
         where
            dimension $t$ ;;
            $k = \lceil \log_2 n \rceil$ ;;
            $A =$ if index.$d > n$ then $1$ else index.$d$ ;;
            $B = A$ fby.$t$ leftchild.$d$ $B$ $*$ rightchild.$d$ $B$ ;;
         end
   end
```

Experiments with a hand-coded C++ multi-threaded program mimicking this structure have shown that we can compute $10^6!$ using this method, while the iterative approach completely breaks down.

[1]For an introduction to algorithmic styles, see *Algorithms*, by Dasgupta, Papadimitriou and Vazirani [24].

7.3.2 Matrix multiplication, part 1

Assume that $n = 2^k$ for some $k \geqslant 0$. Given two matrices, we can multiply them in $\mathcal{O}(n^3)$ time, counting recursive calls. Each non-trivial call branches into eight other calls.

$$\begin{aligned} XY &= \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} \\ &= \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix} \end{aligned}$$

This problem can be readily transformed into a two-dimensional bottom-up calculation, which can then be parallelized for a variety of computer architectures. The key is that the blocking structure for a matrix must be arranged hierarchically, as is shown here for two levels:

$$\left[\begin{array}{cc|cc} A_A & A_B & B_A & B_B \\ A_C & A_D & B_C & B_D \\ \hline C_A & C_B & D_A & D_B \\ C_C & C_D & D_C & D_D \end{array}\right] \left[\begin{array}{cc|cc} E_E & E_F & F_E & F_F \\ E_G & E_H & F_G & F_H \\ \hline G_E & G_F & H_E & H_F \\ G_G & G_H & H_G & H_H \end{array}\right]$$

Below, we show the blocking structure for the elements in a $2^3 \times 2^3$ array:

$$\left[\begin{array}{rr|rr||rr|rr} 0 & 1 & 4 & 5 & 16 & 17 & 20 & 21 \\ 2 & 3 & 6 & 7 & 18 & 19 & 22 & 23 \\ \hline 8 & 9 & 12 & 13 & 24 & 25 & 28 & 29 \\ 10 & 11 & 14 & 15 & 26 & 27 & 30 & 31 \\ \hline\hline 32 & 33 & 36 & 37 & 48 & 49 & 52 & 53 \\ 34 & 35 & 38 & 39 & 50 & 51 & 54 & 55 \\ \hline 40 & 41 & 44 & 45 & 56 & 57 & 60 & 61 \\ 42 & 43 & 46 & 47 & 58 & 59 & 62 & 63 \end{array}\right]$$

To create the TransLucid solution, we first define some needed routines.

```
$knext.d.k\ X = X$ @ $[d : d + 2^{k-1}]$ ;;
$kprev.d.k\ X = X$ @ $[d : d - 2^{k-1}]$ ;;
$combine1.d_1.k\ A\ B = Z$
where
  $Z \mid [d_1 : 0..2^{k-1} - 1] = A$ ;;
  $Z \mid [d_1 : 2^{k-1}..2^k - 1] = kprev.d_1.k\ B$ ;;
end
$combine2.d_1.d_2.k\ A\ B\ C\ D = Z$
where
  $Z \mid [d_1 : 0..2^{k-1} - 1, d_2 : 0..2^{k-1} - 1] = A$ ;;
  $Z \mid [d_1 : 2^{k-1}..2^k - 1, d_2 : 0..2^{k-1} - 1] = kprev.d_1.k\ B$ ;;
  $Z \mid [d_1 : 0..2^{k-1} - 1, d_2 : 2^{k-1}..2^k - 1] = kprev.d_2.k\ C$ ;;
  $Z \mid [d_1 : 2^{k-1}..2^k - 1, d_2 : 2^{k-1}..2^k - 1] = kprev.d_1.k\ (kprev.d_2.k\ D)$ ;;
end
```

The TransLucid solution reflects exactly the structure of the diagram.

```
$mult.d_1.d_2.k\ X\ Y =$
  if $k \equiv 0$ then $X * Y$
  else $combine2.d_1.d_2.k\ (AE + BG)(AF + BH)(CE + DG)(CF + DH)$
  where
    $mm = mult.d_1.d_2.(k-1)$ ;;
    $AE = mm\ A\ E$ ;;
    $BG = mm\ B\ G$ ;;
    $AF = mm\ A\ F$ ;;
    $BH = mm\ B\ H$ ;;
    $CE = mm\ C\ E$ ;;
    $DG = mm\ D\ G$ ;;
    $CF = mm\ C\ F$ ;;
    $DH = mm\ D\ H$ ;;
    $A = X$ ;;
    $B = knext.d_1.k\ X$ ;;
    $C = knext.d_2.k\ X$ ;;
    $D = knext.d_2.k\ \big(knext.d_1.k\ X\big)$ ;;
    $E = Y$ ;;
    $F = knext.d_1.k\ Y$ ;;
    $G = knext.d_2.k\ Y$ ;;
    $H = knext.d_2.k\ \big(knext.d_1.k\ Y\big)$ ;;
  end
```

A bottom-up solution can be derived by looking at the problem from an indexical point of view:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

The bottom-up solution uses four local dimensions, i, j, ℓ and t. First a 3-dimensional cube varying in dimensions i, j and ℓ is constructed. Then, at each t-step, this cube is collapsed by a factor of $2 \times 2 \times 2 = 8$, while the square varying in dimensions d_1 and d_2 grows by a factor of $2 \times 2 = 4$.

```
$mult2.d_1.d_2.k\ X\ Y = W$ @ $[t : k]$
where
  dimension $i, j, \ell, t$ ;;
  $Z = X$ @ $[i : \#d_1, \ell : \#d_2] * Y$ @ $[\ell : \#d_1, j : \#d_2]$ ;;
  $W = Z$ fby.$t$ $reduce$ @ $[i : 2 * \#i, j : 2 * \#j, \ell : 2 * \#\ell]$ ;;
  $reduce = combine2.d_1, d_2.$(index.$t$) $U$ (next.$i$ $U$) (next.$j$ $U$) (next.$i$ next.$j$ $U$) ;;
  $U = W +$ next.$\ell$ $W$ ;;
end
```

7.3.3 Matrix multiplication, part 2

Assume that $n = 2^k$ for some $k \geqslant 0$. Given two matrices, we can multiply them using Strassen's algorithm in $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81})$ time, counting recursive calls. Each non-trivial call branches into seven—as opposed to eight—other calls.

$$\begin{aligned} XY &= \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} \\ &= \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix} \end{aligned}$$

where

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ P_3 &= (C + D)E \\ P_4 &= D(G - E) \\ P_5 &= (A + D)(E - H) \\ P_6 &= (B - D)(G + H) \\ P_7 &= (A - C)(E + F) \end{aligned}$$

The TransLucid recursive solution follows exactly the definition.

```
$mult3.d_1.d_2.k\ X\ Y =$
  if $k \equiv 0$ then $X * Y$
  else $combine2.d_1.d_2.k\ (P_5 + P_4 - P_2 + P_6)(P_1 + P_2)(P_3 + P_4)(P_1 + P_5 - P_3 - P_7)$
  where
    $mm = mult3.d_1.d_2.(k - 1)$ ;;
    $P_1 = mm\ A\ (F - H)$ ;;
    $P_2 = mm\ (A + B)\ H$ ;;
    $P_3 = mm\ (C + D)\ E$ ;;
    $P_4 = mm\ D\ (G - E)$ ;;
    $P_5 = mm\ (A + D)\ (E - H)$ ;;
    $P_6 = mm\ (B - D)\ (G - H)$ ;;
    $P_7 = mm\ (A - C)\ (E + F)$ ;;
    $A = X$ ;;
    $B = knext.d_1.k\ X$ ;;
    $C = knext.d_2.k\ X$ ;;
    $D = knext.d_2.k\ \big(knext.d_1.k\ X\big)$ ;;
    $E = Y$ ;;
    $F = knext.d_1.k\ Y$ ;;
    $G = knext.d_2.k\ Y$ ;;
    $H = knext.d_2.k\ \big(knext.d_1.k\ Y\big)$ ;;
  end
```

Like for the $\mathcal{O}(n^3)$ solution, a bottom-up solution could be generated, but at the cost of additional memory. To go down a level, we would need to keep A, D, E and H, and to create $F-H$, $A+B$, $C+D$, $G-E$, $A+D$, $E-H$, $B-D$, $G+H$, $A-C$ and $E+F$. There are therefore 14 submatrices generated from 8. As a result, the bottom-up solution would require $\frac{7}{2} = 1.75$ times the space for every level.

Possibly, a hybrid solution using Strassen for the first 2 or 3 layers, then the $\mathcal{O}(n^3)$ solution for the lower layers, would be best. For example, if we wish to multiply two $1024 \times 1024 = 2^{10} \times 2^{10}$ matrices, we could apply Strassen's algorithm for three levels, then the ordinary algorithm for the $343 = 7^3$ smaller ($128 \times 128 = 2^7 \times 2^7$) matrices.

7.3.4 Merge sort

Assume that $n = 2^k$ for some $k \geqslant 0$. Suppose we wish to sort an array of n elements. Using merge sort, we divide the array into two subarrays of 2^{k-1} elements, sort these, then merge the two resulting arrays. Here is the TransLucid program:

```
$msort.d.k\ X =$
  if $k \equiv 0$ then $X$
  else $mer.d.(k-1)\ A\ B$
  where
    $A = X\ ;;$
    $B = knext.d.k\ X\ ;;$
    $mer.d.k\ X\ Y =$ next.$d\ z'$
    where
      dimension $x, y, z\ ;;$
      $T = [x : 0, y : 0, z : 0]$ fby.$d$
          if $x' \equiv 2^k$
            then $\big[x : x', y : y' + 1, z : Y$ @ $[d : y']\big]$
          elsif $y' \equiv 2^k$
            then $\big[x : x' + 1, y : y', z : X$ @ $[d : x']\big]$
          elsif $X$ @ $[d : x'] \leqslant Y$ @ $[d : y']$
            then $\big[x : x' + 1, y : y', z : X$ @ $[d : x']\big]$
            else $\big[x : x', y : y' + 1, z : Y$ @ $[d : y']\big]\ ;;$
          $x' =$ #$x$ @ $T\ ;;$
          $y' =$ #$y$ @ $T\ ;;$
          $z' =$ #$z$ @ $T\ ;;$
    end
  end
```

The bottom-up solution is derived directly:

```
$msort2.d.k\ X = Z$ @ $[t : k]$
where
  dimension $t\ ;;$
  $Z = X$ fby $mer.d.($#$t)\ Z\ \big(knext.d.($index.$t)\ Z\big)\ ;;$
  $mer.d.k\ X\ Y =$ next.$d\ z'$
  where
    dimension $x, y, z\ ;;$
    $T = [x : 0, y : 0, z : 0]$ fby.$d$
        if $x' \equiv 2^k$
          then $\big[x : x', y : y' + 1, z : Y$ @ $[d : y']\big]$
        elsif $y' \equiv 2^k$
          then $\big[x : x' + 1, y : y', z : X$ @ $[d : x']\big]$
        elsif $X$ @ $[d : x'] \leqslant Y$ @ $[d : y']$
          then $\big[x : x' + 1, y : y', z : X$ @ $[d : x']\big]$
          else $\big[x : x', y : y' + 1, z : Y$ @ $[d : y']\big]\ ;;$
        $x' =$ #$x$ @ $T\ ;;$
        $y' =$ #$y$ @ $T\ ;;$
        $z' =$ #$z$ @ $T\ ;;$
  end
end
```

7.3.5 Fast Fourier transform

The fast Fourier transform (FFT) is one of the most important algorithms, and forms the basis for much signal processing. Here is a recursive definition in TransLucid.

```
$fft.d.kX =$
  if $k \equiv 0$ then $X$ else $combine1.d.k\ A'\ B'$
  where
    $A' = A + \omega.k * B$ ;;
    $B' = A - (knext.d.k\ \omega.k) * B$ ;;
    $A = fft.d.(k-1)\ X$ ;;
    $B = fft.d.(k-1)\ \big(knext.d.k\ X\big)$ ;;
    $\omega.k =$ the complex $2^k$-th roots of unity
  end
```

The bottom-up version has a similar structure.

```
$fft2.d.kX = Y$ @ $[t : k]$
where
  dimension $t$ ;;
  $Y = X$ fby.$t$ $combine1.d.($index.$t)\ A'\ B'$
  $A' = Y + \omega.($index.$t) * \big(knext.d.($index.$t)\ Y\big)$ ;;
  $B' = Y - knext.d.($index.$t)\ \big(\omega.($index.$t) * Y\big)$ ;;
  $\omega.k =$ the complex $2^k$-th roots of unity
end
```

This last solution would lead naturally to the butterfly network, which is a standard hardware implementation of FFT.

7.4 Dynamic programming algorithms

Like divide-and-conquer, dynamic programming is a method for solving problems that consist in generating subproblems and solving them in turn. However, each subproblem may well be a size comparable to that of the original problem, and the generation of subproblems often leads to significant repetition. As a result, it is common to use memoization to ensure that subproblems are only computed once. Once again, bottom-up calculation can be very effective.

We look at one example.

7.4.1 Edit distance

The *edit distance* between two words is computed by counting the minimum number of deletions, insertions or substitutions required to go from one word to the other. By looking at the i-th prefix of the first word and the j-th prefix of the second word, there are three possible subproblems, whose results must be brought together, giving the following code:

```
for $i = 0..m-1$
  $E(i,0) = i$
for $j = 0..n-1$
  $E(0,j) = j$
for $i = 1..m-1$
  for $j = 1..n-1$
    $E(i,j) = \min\big\{E(i-1,j)+1,\ E(i,j-1)+1,\ E(i-1,j-1)+\mathrm{diff}(i,j)\big\}$
return $E(m,n)$
```

Here is the TransLucid program.

```
$edit.i.m.n\ A\ B = E$ @ $[i : m-1, j : n-1]$
  where
    dimension $j$ ;;
    $E \mid [i = 0..m-1] =$ #$i$ ;;
    $E \mid [j = 0..n-1] =$ #$j$ ;;
    $E = min\ \{$prev.$i\ E + 1,$ prev.$j\ E + 1,$ prev.$i$ prev.$j\ E + diff\}$ ;;
    $diff =$ if $A \not\equiv (B$ @ $[i :$ #$j])$ then $0$ else $1$ ;;
  end
```

This code can be implemented so that it only needs linear space. Here is an (almost) C++ solution. (Assume $m \equiv n$.)

```
unsigned
edit (const unsigned n, A[n], B[n])
{
  unsigned E[n];
  for (unsigned i = 0; i != n; ++i)
     E[i] = i;
  for (unsigned j = 1; j != n; ++j)
  {
    unsigned d = j;
    for (unsigned i = 1; i != n; ++i)
      d = E[i] = min(d, E[i], diff[i][j]);
  }
  return E[n-1];
}
```

7.5 Conclusions

Both for divide-and-conquer and dynamic programming algorithms, the generation of bottom-up solutions with minimal additional memory consumption is a very important goal for TransLucid programming and implementation. Ideally, one would write a recursive function and the bottom-up solution would be automatically generated. In practice, a combination of automatic and manual steps is needed. More experimentation is needed, not only to produce these bottom-up solutions automatically, but also to produce tools that would assist in the manual generation of such solutions.

The necessary code generation for TransLucid requires much work, since we wish to ensure that all entities being manipulated to produce code for TransLucid be themselves understandable from the point of view of TransLucid. This means that the implementation techniques such as memoize-and-reset functions and bottom-up computations need to be translatable back to TransLucid. To do this requires understanding imperative programming, which is discussed in the next chapter.

Chapter 8

Putting Control on the Index

The TransLucid language is declarative. A question thus arises: Can Cartesian programming deal with the complexities of “real programming”, namely with side-effects and imperative programming? We examine these issues in this chapter, and demonstrate that the Cartesian approach helps clarify some issues.

8.1 Side-effects in functions

TransLucid is designed to be used as a coordination language, in which constants and data functions are defined in the host language. We have been supposing up to now that these constants and data functions are stateless, and that the only semantics of interest is in the behavior of TransLucid. However, it is quite possible, maybe even desirable, for these constants and functions to have state. This raises the question, Is it possible for this to make sense in a declarative manner? The answers we will provide in this chapter will rely on the Cartesian framework. Rather than focusing on the semantics of the side-effect itself, we consider the very concept of side-effect, and come up with some basic conclusions.

First, we should note that side-effects are unavoidable in computing. In fact, if there were no side-effects, computing would be useless: should one put a computer in a locked chamber with no connection to the outside world but the electrical supply, then that computer would be completely useless! Sooner or later, something needs to be printed, displayed on a screen, written to disk or sent over a network, or some actuator needs to be activated.

One possible answer is to state that since a TransLucid system is a reactive system, it can be left to the calling environment to determine if these kinds of action need to be performed, and to do so should this be the case. Clearly, this kind of choice is perfectly reasonable for some applications, but is not satisfactory in general, as the power of the host language and the underlying computer system would not be available.

So what is a side-effect? It is a change of some external state, not accessible to the TransLucid system. Let us consider as example that it is page 232 of a book being printed. Then if page 232 is to be printed, it must be printed exactly once. Furthermore, all of the pages preceding page 232 have to printed before printing it.

What we can retain from this example is that a side-effect can only take place *once* and that it may depend on other side-effects having taken place prior to it taking place. How can we ensure this sort of thing in TransLucid?

The answer is surprisingly simple. An external function f which might provoke a side-effect is declared in the header as such, and a set of dimensions $d_1, \ldots, d_n$ is declared to be variant with respect to the execution of f. For example:

$$\texttt{stateful}\ f\ \texttt{dimensions}\ d_1, \ldots, d_n\ ;;$$

Then, for any point in $d_1 \times \cdots \times d_n$ space, f may be evaluated exactly once. Should f be evaluated at point $\kappa = [d_1 : c_1, \ldots, d_n : c_n]$, its arguments and return value must be cached (see Chapter 4). If f is called a second time at κ, then the cache must check to ensure that the same arguments are passed; if they are, the cached value is returned, otherwise `sp⟨state⟩`.

In addition to being simple, this answer has the advantage that it is fully under the control of the programmer. Should the programmer decide that no control over side-effects is necessary, then the declaration of f refers to no dimensions of variance. Furthermore, it allows different kinds of side-effect to be subject to different sets of dimensions.

For the dependencies, two techniques are appropriate. The first introduces the ";" operator: $E_1; E_2$ returns the result of evaluating E_2, but only initiates E_2 once the evaluation of E_1 is finished. Should E_1 and E_2 both include side-effects, then the side-effects of E_1 take place before those of E_2.

The second method is to introduce a new kind of declaration, this time added to the equation set, implying a dependency between different points in the Cartesian space. Dependencies would be defined by bestfitting, like for equations. A guard g for a dependency for variable x would look like this:

$$B_g \;\leftarrow\; E_g$$

where E_g would have to evaluate to a tuple. The idea is that if the current context were contained inside the region defined by B_g, then x would first have to be evaluated at the context defined by E_g.

8.2 Side-effects in objects

In a language such as `C++`, side-effects do not just take place through function calls, but also through the use of class methods. Can these too be added to TransLucid? Conceptually, the answer is yes, using the same techniques as above, with an additional dimension for the object in question.

In practice, this would be more difficult, as one would need to add class declarations to TransLucid, complete with the typing of the methods. Should this work have been done and the interpreter adjusted accordingly, then we could refine the above techniques by distinguishing between `set` and `get` methods. Since `get` methods are equivalent to side-effect-free functions, if an object is *clonable*, then uses of `get` methods would not need to be cached: caching the object would suffice, since one could recall `get` methods at will.

It would probably be safer to only work with objects that respected some clear interface, which might wrap around other more general objects, in such a way as to guarantee that the objects act as TransLucid hyperdatons, with synchronous semantics (see §6.7).

8.3 Imperative programming

To understand side-effects declaratively required the use of a set of dimensions in which were registered these effects. For an imperative program P, the same process takes place, but with a fixed set of dimensions, corresponding to the different accessible points in P, along with the iteration counts of loops belonging to P. The approach corresponds closely to those used for modern compilers [22], which use static single-assignment form to register all of the different assignments to a specific variable, and which attempt to treat as a single sequence all those instructions executed between two I/O operations.

We present a simple imperative language with blocks, conditionals, loops, function calls, dynamic exceptions and side-effects. We then label the different positions in a program in order for it to be translated into TransLucid. We assume that the semantics of an imperative program is the execution, in the correct order, of the chain of side-effects within the program, followed by the evaluation of the return result.

8.3.1 Grammar

The grammar for the language is given below. All statements can be labeled, and the label can be used as a context, as in x `@` $[\ell]$, meaning the value of x at label ℓ. Loops must be exited with a labeled exit, and the next iteration started with a labeled continue. For blocks that are not loops, labeled exits and continues are equivalent.

In the grammar, enumerations with ... may have no (0) occurrences.

$$
\begin{array}{rcl}
P & ::= & F \\
F & ::= & \texttt{main (}\ x\ \texttt{,}\ \ldots\ \texttt{,}\ x\ \texttt{)}\ B \\
B & ::= & \{\ LS\ \ldots\ LS\ \} \\
LS & ::= & \ell\ \texttt{:}\ S \\
 & | & S \\
S & ::= & E\ \texttt{;} \\
 & | & \texttt{exit}\ \ell\ \texttt{;} \\
 & | & \texttt{continue}\ \ell\ \texttt{;} \\
 & | & B \\
 & | & \texttt{loop}\ B \\
 & | & \texttt{if (}\ E\ \texttt{)}\ B\ B \\
E & ::= & c \\
 & | & x \\
 & | & x\ \texttt{@}\ C \\
 & | & x\ \texttt{:=}\ E \\
 & | & op\ (E, \ldots, E) \\
 & | & op_S\ (E, \ldots, E) \\
C & ::= & \texttt{[}\ L, \ldots, L\texttt{]} \\
L & ::= & \ell \mid \ell\text{-}n
\end{array}
$$

where

B is a *block*;

C is a *context*;

E is a *expression*;

F is a *function*;

P is a *program*;

S is a *statement*;

LS is a *labelled statement*;

c is a *constant*;

ℓ is a *label*;

x is a *variable*;

op is an *operator*;

op_S is an operator with side-effect.

The @ operator is used to refer to the value of an operator in a previous position. Normally one will only use a single label, but should this position be within a loop, then one can write $\ell - n$ to designate the value n iterations previous from the current one in the loop labelled by ℓ.

The labels are needed to exit or continue blocks, and are mostly used in loops. For example:

```
main ()
{
  x := 0;
  L1: loop
      {
        x := x+1 ;
        if (x>5) { exit L1; } { }
        if (x>4) { continue L1; } {}
        x := x+1 ;
      }
}
```

The `exit L1;` means to exit the loop labelled `L1`. The `continue L1;` means to start the next iteration of the loop labelled `L1`.

8.3.2 Tagging blocks and statements

Each block and statement in a program has a unique tag. This section describes the tagging process. The basic idea is that the statements in a block are numbered, and the positions *between* the statements are numbered as well. Suppose we have a block with n statements:

$$B = \{\ LS_0\ \ldots\ LS_{n-1}\ \}$$

Then there are $n + 1$ positions, corresponding to the beginning (1 position), and the semicolons and block close braces (n positions, including the end):

$$B = \{\ p_0\ LS_0\ p_1\ \ldots\ p_{n-1}\ LS_{n-1}\ p_n\ \}$$

If there were only one block in a system, then the positions could simply be tagged with $0, 1, \ldots, n$. However, the blocks appear in a hierarchical arrangement, so we need to keep track of this hierarchy. For example, in program:

```
main ()
{
  x := 1 ;
  y:=3 ;
  print (x) ;
  print (y) ;
}
```

the tagging would be like this:

```
main ()
{
  [main:0]
  x := 1;
  [main:1]
  y := 3;
  [main:2]
  print (x);
  [main:3]
  print (y);
  [main:4]
}
```

while in program

```
main ()
{
  x := 1 ;
  if (x>1)
  {
    y:=3 ;
  }
  {
    y:=4 ;
  }
  print (y) ;
}
```

the tagging would give:

```
main ()
{
  [main:0]
  x := 1;
  [main:1]
  if (x > 1)
  {
    [main:1,iftrue:0]
    y := 3;
    [main:1,iftrue:1]
  }
  {
    [main:1,iffalse:0]
    y := 4;
    [main:1,iffalse:1]
  }
  [main:2]
  print (y);
  [main:3]
}
```

The tagging takes place by traversing the parse tree, and is defined structurally. There are five functions.

- $\mathcal{T}_0(\mathit{function})$ is the top-level function called to start the process.
- $\mathcal{T}_1(\mathit{statement}, \mathit{tag})$ is the function called to tag a statement. The *tag* is a list.
- $\mathcal{T}_2(\mathit{block}, \mathit{tag}, \mathit{prefix})$ is the function called to tag a block. The *tag* is a list and the *prefix* is a key used to identify the current block.
- $\mathcal{T}_3(\mathit{expression}, \mathit{tag}, \mathit{prefix})$ is the function called to tag an expression.
- $\mathcal{T}_4(\mathit{expression}, \mathit{tag}, \mathit{prefix})$ is a subsidiary function of $\mathcal{T}_3$.

$$
\begin{array}{rcl}
\mathcal{T}_0(F_0\ \ldots\ F_{n-1}) & = & \mathcal{T}_0\big(F_0\big)\ \ldots\ \mathcal{T}_0(F_{n-1}) \\
\mathcal{T}_0\big(x\ \texttt{(}\ x_0\ \texttt{,}\ \ldots\ \texttt{,}\ x_{n-1}\ \texttt{)}\ B\big) & = & x\ \texttt{(}\ x_0\ \texttt{,}\ \ldots\ \texttt{,}\ x_{n-1}\ \texttt{)}\ \mathcal{T}_2(B,[],x)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}_1(\ell : S, \mathcal{L}) & = & \ell : \mathcal{T}_1(S,\mathcal{L}) \\
\mathcal{T}_1(E,\mathcal{L}) & = & \mathcal{T}_3\big(E,\mathcal{L},[]\big) \\
\mathcal{T}_1(\texttt{exit}\ \ell,\mathcal{L}) & = & \texttt{exit}\ \ell \\
\mathcal{T}_1(\texttt{continue}\ \ell,\mathcal{L}) & = & \texttt{continue}\ \ell \\
\mathcal{T}_1(B,\mathcal{L}) & = & \mathcal{T}_2(B,\mathcal{L},\texttt{block}) \\
\mathcal{T}_1(\texttt{loop}\ B,\mathcal{L}) & = & \mathcal{T}_2(B,\mathcal{L},\texttt{loop}) \\
\mathcal{T}_1(\texttt{if (}E\texttt{)}\ B_1\ B_2,\mathcal{L}) & = & \texttt{if (}E\texttt{)} \\
 & & \mathcal{T}_2(B_1,\mathcal{L},\texttt{iftrue}) \\
 & & \mathcal{T}_2(B_2,\mathcal{L},\texttt{iffalse})
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}_2\big(\{LS_0\ldots LS_{n-1}\},\mathcal{L},\mathcal{X}\big) & = & \{ \\
 & & \quad \texttt{[}\mathcal{L}\texttt{, }\mathcal{X}\texttt{:}0\texttt{]} \\
 & & \quad \mathcal{T}_1\big(LS_0,[\mathcal{L}\cdot\mathcal{X}:0]\big) \\
 & & \quad \texttt{[}\mathcal{L}\texttt{, }\mathcal{X}\texttt{:}1\texttt{]} \\
 & & \quad \ldots \\
 & & \quad \texttt{[}\mathcal{L}\texttt{, }\mathcal{X}\texttt{:}n-1\texttt{]} \\
 & & \quad \mathcal{T}_1\big(LS_n-1,[\mathcal{L}\cdot\mathcal{X}:n-1]\big) \\
 & & \quad \texttt{[}\mathcal{L}\texttt{, }\mathcal{X}\texttt{:}n\texttt{]} \\
 & & \} \\
\mathcal{T}_3(E,\mathcal{L},\mathcal{X}) & = & [\mathcal{L}\cdot\mathcal{X}b]\ \ \mathcal{T}_4(E,\mathcal{L},\ \mathcal{X})\ \ [\mathcal{L}\cdot\mathcal{X}e] \\
\mathcal{T}_4(c,\mathcal{L},\mathcal{X}) & = & c \\
\mathcal{T}_4(x,\mathcal{L},\mathcal{X}) & = & x \\
\mathcal{T}_4(x\ \texttt{@}\ C,\mathcal{L},\mathcal{X}) & = & x\ \texttt{@}\ C \\
\mathcal{T}_4(x := E,\mathcal{L},\mathcal{X}) & = & x := \mathcal{T}_3(E,\mathcal{L},\mathcal{X}0) \\
\mathcal{T}_4\big(op(E_0,\ldots,E_{n-1}),\mathcal{L},\mathcal{X}\big) & = & op\Big(\mathcal{T}_3(E_0,\mathcal{L},\mathcal{X}0),\ldots\mathcal{T}_3(E_{n-1},\mathcal{L},\mathcal{X}(n-1))\Big)
\end{array}
$$

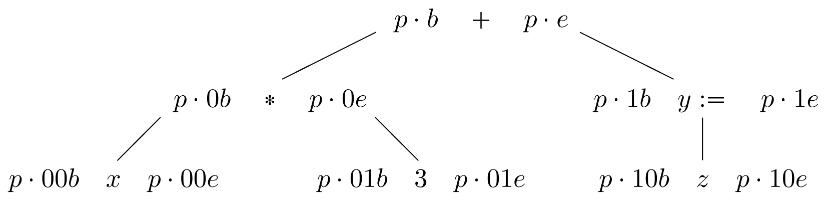
8.3.3 Interpreting expressions

The objective of the translation of an imperative program into TransLucid is to generate a set of TransLucid equations that track the data flow of all variables. Like in `C`, each statement is assumed to have a value, which is the value of the last evaluated expression. Because the expressions can modify the values of variables, we need to keep track of state during the evaluation of expressions.

For the expressions, we explain the process using an example, namely:

$$p \quad (x * 3) + (y := z); \quad p'$$

which is found at current position is p and whose next position is p'. Then the tagged expression corresponds to the following figure:


$p \cdot b$ $+$ $p \cdot e$
$p \cdot 0b$ $*$ $p \cdot 0e$
$p \cdot 1b$ $y :=$ $p \cdot 1e$
$p \cdot 00b$ x $p \cdot 00e$
$p \cdot 01b$ 3 $p \cdot 01e$
$p \cdot 10b$ z $p \cdot 10e$

Therefore, within the expression, there are 12 control positions: $p \cdot b$, $p \cdot 0b$, $p \cdot 00b$, $p \cdot 00e$, $p \cdot 01b$, $p \cdot 01e$, $p \cdot 0e$, $p \cdot 1b$, $p \cdot 10b$, $p \cdot 10e$, $p \cdot 1e$, $p \cdot e$.

The current value of the expression is kept track of in the variable called `cur`, for *current*. All evaluation of expressions is assumed to be leftmost, depthfirst, to simplify the generation of the rules. The generated equations for the above expression are:

$$S \mathbin{\mathtt{|}} [\mathtt{P} : p'] = S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot e];;$$

$$\begin{aligned}
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot b] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p];;\\
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot e] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 1e];;\\
S \mathbin{\mathtt{|}} [\mathtt{var} : \mathtt{cur}, \mathtt{P} : p \cdot e] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 0e] + S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 1e];;
\end{aligned}$$

$$\begin{aligned}
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 0b] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot b];;\\
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 0e] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 01e];;\\
S \mathbin{\mathtt{|}} [\mathtt{var} : \mathtt{cur}, \mathtt{P} : p \cdot 0e] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 00e] * S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 01e];;
\end{aligned}$$

$$\begin{aligned}
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 00b] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 0b];;\\
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 00e] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 00b];;\\
S \mathbin{\mathtt{|}} [\mathtt{var} : \mathtt{cur}, \mathtt{P} : p \cdot 00e] &= S \mathbin{\mathtt{@}} [\mathtt{var} : x, \mathtt{P} : p \cdot 00b];;
\end{aligned}$$

$$\begin{aligned}
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 01b] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 00e];;\\
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 01e] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 01b];;\\
S \mathbin{\mathtt{|}} [\mathtt{var} : \mathtt{cur}, \mathtt{P} : p \cdot 01e] &= 3;;
\end{aligned}$$

$$\begin{aligned}
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 1b] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 0e];;\\
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 1e] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 10e];;\\
S \mathbin{\mathtt{|}} [\mathtt{var} : y, \mathtt{P} : p \cdot 1e] &= S \mathbin{\mathtt{@}} [\mathtt{var} : \mathtt{cur}, \mathtt{P} : p \cdot 10e];;
\end{aligned}$$

$$\begin{aligned}
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 10b] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 1b];;\\
S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot 10e] &= S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot 10b];;\\
S \mathbin{\mathtt{|}} [\mathtt{var} : \mathtt{cur}, \mathtt{P} : p \cdot 10e] &= S \mathbin{\mathtt{@}} [\mathtt{var} : z, \mathtt{P} : p \cdot 10b];;
\end{aligned}$$

Of course, the above equations do not express how $S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot b]$ is defined, nor how $S \mathbin{\mathtt{@}} [\mathtt{P} : p \cdot e]$ will be used. Hence, for every node, three equations are generated:

1. The equation defining the state of the system prior to commencing this part of the computation. This is the state of the system at the end of the left sibling if it exists, otherwise the state of the system at the beginning of the parent node.

2. The equation defining the current node. This defines variable `cur`, if the current node is not an assignment, otherwise it defines the variable named in the assignment.

3. The equation defining the rest of the system. This is the state of the system at the beginning of the current node, if it is for a variable or a constant, otherwise it is the state of the system at the end of the rightmost child node.

8.3.4 Interpreting expression context changes

The syntax for expressions includes a line for context changes.

$$\begin{array}{rcl} E & ::= & \ldots \\ & | & x \mathtt{@} C \\ C & ::= & \mathtt{[}\, L, \ldots, L\mathtt{]} \\ L & ::= & \ell \mid \ell\text{–}n \end{array}$$

The change of context allows one to refer to the value of a variable, including `cur`, at a previous instant. Exactly one of the labels must be an ordinary label ℓ_p for an ordinary position p, and must be for a position which is definitely reached before reaching the current position; this label must be placed first. For the other labels, the labels must be for loops which include the current position. The translation into TransLucid for the expression:

$$x \mathtt{@} [\ell_p, \ell_1 - 2, \ell_2 - 3, \ell_3 - 4]$$

simply becomes:

$$S \mathtt{@} [\mathtt{var} : x, \mathtt{P} : \ell_p, \ell_1 : \mathtt{\#}\ell_1 - 2, \ell_2 : \mathtt{\#}\ell_2 - 3, \ell_3 : \mathtt{\#}\ell_3 - 4]$$

8.3.5 Interpreting side-effects

The syntax for expressions includes a line for operators with side-effects.

$$\begin{array}{rcl} E & ::= & \ldots \\ & | & op_S\ (E, \ldots, E) \end{array}$$

To properly implement a program means to ensure that the side-effects to be created should be created in the right order, then that the final return value be evaluated. To do this, we must introduce variable, called `last`, to keep track of the position in which the last side-effect was effected.

Suppose that there is a unary operator `print`, which will print a value to standard output, and that it then returns a status value. Suppose that we are encoding the statement

$$p \quad \mathtt{print}\,(p \cdot b \quad E \quad p \cdot e) \quad p'$$

Then the translation will be:

$$\begin{array}{r} S \mathbin{\mathtt{|}} [\mathtt{P} : p'] = S \mathtt{@} [\mathtt{P} : p \cdot e];; \\ S \mathbin{\mathtt{|}} [\mathtt{P} : p \cdot b] = S \mathtt{@} [\mathtt{P} : p];; \\ S \mathbin{\mathtt{|}} [\mathtt{var} : \mathtt{cur}, \mathtt{P} : p'] = S \mathtt{@} [\mathtt{var} : \mathtt{last}, \mathtt{P} : p'];; \\ S \mathbin{\mathtt{|}} [\mathtt{var} : \mathtt{last}, \mathtt{P} : p'] = S \mathtt{@} [\mathtt{var} : \mathtt{last}, \mathtt{P} : p \cdot e];\ \mathtt{print}\ \big(S \mathtt{@} [\mathtt{var} : \mathtt{cur}, \mathtt{P} : p \cdot e]\big);; \end{array}$$

8.3.6 Interpreting labels

A label is used to exit from a block or a loop or to continue on with the next iteration of a loop. A label can also be referred to in the evaluation of an expression in order to refer to the value a variable held in a previous location in the program, thereby alleviating the need for temporary variables.

In our approach, the way we handle `exit`'s and `continue`'s is to consider them to be special kinds of value. Taking this approach radically simplifies their tracking through conditional blocks.

Consider the following exit statement at position p.

$$p \quad \texttt{exit}\,\ell; \quad p'$$

Its translation is simply:

$$\begin{gathered} S \mid [\texttt{P} : p'] = S \mathbin{\texttt{@}} [\texttt{P} : p];; \\ S \mid [\texttt{var} : \texttt{cur}, \texttt{P} : p'] = \texttt{exit}\ \big(\ell, S \mathbin{\texttt{@}} [\texttt{var} : \texttt{cur}, \texttt{P} : p]\big);; \end{gathered}$$

Similarly, consider the following continue statement at position `P`:

$$p \quad \texttt{continue}\,\ell; \quad p'$$

Its translation is simply:

$$\begin{gathered} S \mid [\texttt{P} : p'] = S \mathbin{\texttt{@}} [\texttt{P} : p];; \\ S \mid [\texttt{var} : \texttt{cur}, \texttt{P} : p'] = \texttt{continue}\ \big(\ell, S \mathbin{\texttt{@}} [\texttt{var} : \texttt{cur}, \texttt{P} : p]\big);; \end{gathered}$$

In other words, these are handled as if they are expressions.

8.3.7 Interpreting blocks

A block is interpreted by chaining all of the statements therein together. However, because of the possibility of `exit` and `continue` values generated from within these statements, checks must be made for these values, so that they can be passed from statement to statement without any further calculations or changes of state taking place. To do this means generating an equation that allows the bypassing of a statement if the current value is already an `exit` or `continue`. For example, suppose there were an expression statement of the form:

$$p \quad E; \quad p'$$

Equations of the following form should be generated:

$$\begin{gathered} S \mid [\texttt{P} : p \cdot b] = S \mathbin{\texttt{@}} [\texttt{P} : p];; \\ \cdots \\ S \mid [\texttt{P} : p'] = S \mathbin{\texttt{@}} [\texttt{P} : p \cdot e];; \end{gathered}$$

We need to add another equation to the last one.

$$\begin{aligned} & S \mid [\texttt{priority} : 1, \texttt{P} : p'] \\ & \quad \texttt{\&}\ \texttt{istype}\langle\texttt{exit}\rangle\ \big(S \mathbin{\texttt{@}} [\texttt{var} : \texttt{cur}, \texttt{P} : p]\big) \vee \texttt{istype}\langle\texttt{continue}\rangle\ \big(S \mathbin{\texttt{@}} [\texttt{var} : \texttt{cur}, \texttt{P} : p]\big) \\ & \quad = S \mathbin{\texttt{@}} [\texttt{P} : p];; \end{aligned}$$

In other words, if there is an `exit` or `continue` value, then the statement is effectively skipped.

8.3.8 Interpreting conditional statements

Consider the conditional statement, where we have indicated the key positions:

$$\begin{array}{l} p \\ \texttt{if}\ (p\cdot b \quad E \quad p\cdot e) \\ \{p\cdot \texttt{iftrue}:0 \quad B_T \quad p\cdot\texttt{iftrue}:n_T\} \\ \{p\cdot \texttt{iffalse}:0 \quad B_F \quad p\cdot\texttt{iffalse}:n_F\} \\ p' \end{array}$$

Here is the translation.

$$\begin{aligned} S \mid [\texttt{P}:p\cdot b] &= S\ \texttt{@}\ [\texttt{P}:p];; \\ S \mid [\texttt{P}:p\cdot e] &= \cdots \end{aligned}$$

$$\begin{aligned} S \mid [\texttt{P}:p\cdot \texttt{iftrue}:0] &= S\ \texttt{@}\ [\texttt{P}:p\cdot e];; \\ S \mid [\texttt{P}:p\cdot \texttt{iffalse}:n_T] &= \cdots \end{aligned}$$

$$\begin{aligned} S \mid [\texttt{P}:p\cdot \texttt{iffalse}:0] &= S\ \texttt{@}\ [\texttt{P}:p\cdot e];; \\ S \mid [\texttt{P}:p\cdot \texttt{iffalse}:n_F] &= \cdots \end{aligned}$$

$$\begin{aligned} S \mid [\texttt{P}:p'] = {} & \texttt{if}\ S\ \texttt{@}\ [\texttt{var}:\texttt{cur},\texttt{P}:p\cdot e] \\ & \texttt{then}\ S\ \texttt{@}\ [\texttt{P}:p\cdot\texttt{iftrue}:n_T] \\ & \texttt{else}\ S\ \texttt{@}\ [\texttt{P}:p\cdot\texttt{iffalse}:n_F] \\ & \texttt{fi};; \end{aligned}$$

8.3.9 Interpreting loops

Consider the loop construct starting at position P:

$$p \quad L:\texttt{loop}\ \{p\cdot\texttt{loop}:0 \quad B \quad p\cdot\texttt{loop}:n\} \quad p'$$

The translation requires the generation of three new variables and of a new dimension for each loop, all of which are derived from the label itself.

$$\begin{aligned} L &= p;; \\ S \mid [\texttt{P}:p'] &= S\ \texttt{@}\ \left[\texttt{P}:p\cdot\texttt{loop}:n, L:\texttt{it}_L\ \texttt{@}\ [L:0]\right];; \\ S \mid [\texttt{var}:\texttt{cur},\texttt{P}:p'] &= \texttt{if}\ \texttt{final}_L \equiv \texttt{exit}\ (T,i)\ \texttt{then}\ i\ \texttt{else}\ \texttt{final}_L\ \texttt{fi};; \end{aligned}$$

$$\begin{aligned} \texttt{it}_L = {} & \texttt{if}\ \texttt{istype}\langle\texttt{exit}\rangle\ (\texttt{end}_L)\ \texttt{||} \\ & \quad \texttt{end}_L \equiv \texttt{continue}(L',i), L' \neq L \\ & \texttt{then}\ \texttt{\#}L\ \texttt{else}\ \texttt{it}_L\ \texttt{@}\ [L:\texttt{\#}L+1]\ \texttt{fi};; \\ \texttt{end}_L = {} & S\ \texttt{@}\ [\texttt{var}:\texttt{cur},\texttt{P}:p\cdot\texttt{loop}:n];; \\ \texttt{prev}_L = {} & \texttt{end}_L\ \texttt{@}\ [L:\texttt{\#}L-1];; \\ \texttt{final}_L = {} & \texttt{end}_L\ \texttt{@}\ \left[L:it_L\ \texttt{@}\ [L:0]\right];; \end{aligned}$$

$$\begin{aligned} S \mid [\texttt{P}:p\cdot\texttt{loop}:0] &= S\ \texttt{@}\ [\texttt{P}:p\cdot\texttt{loop}:n, L:\texttt{\#}L-1];; \\ S \mid [\texttt{var}:\texttt{cur},\texttt{P}:p\cdot\texttt{loop}:0] &= \texttt{if}\ \texttt{prev}_L \equiv \texttt{continue}\ (L,i)\ \texttt{then}\ i\ \texttt{else}\ \texttt{prev}_L\ \texttt{fi};; \\ S \mid [\texttt{priority}:1,\texttt{P}:p\cdot\texttt{loop}:0, L:0] &= S\ \texttt{@}\ [\texttt{P}:p];; \end{aligned}$$

8.3.10 Bringing it all together

The semantics of the `main` function, starting at point p and finishing at point p', consists of the following demand:

$$S \;@\; [\mathtt{var} : \mathtt{last}, \mathtt{P} : p']; S \;@\; [\mathtt{var} : \mathtt{cur}, \mathtt{P} : p']$$

In other words, first make sure that all of the side-effects take place, then create the return result.

For this process to work, then the definition for `last` must be given for the starting point p:

$$S \mid [\mathtt{P} : p] = 0;;$$

It is straightforward to add checks to block exits to see if there is an exit or a continue, as well as to add dynamic exceptions, as are found in languages such as `C++`.

More difficult, but still straightforward, is the function call, because a stack of contexts is required. Upon entry in a called function, the `last` entry must keep track of the previous side-effect, wherever it might have occurred in the calling stack.

A translator for this language to TransLucid has already been written. Note that using this approach, a completely declarative semantics for imperative programs can be generated, and that the generated set of equations can then be examined using dataflow analysis.

8.4 Conclusions

In addition to allowing us to explore imperative programming from an indexical point of view for code generation purposes, the main aim of this chapter is to show that TransLucid can serve as intermediate language for a wide variety of languages, including procedural ones, and that this translation process can encourage new developments in the original langUge. In fact, a number of paradigms can easily be understood from the multidimensional perspective.

- In languages with functions and no objects, only one dimension is used to distinguish methods: the function name.
- In object-oriented programming, there are two dimensions: the name of the method and the `receiver` object of the message.
- In subjective programming [86], there are three dimensions: the name of the method, the `receiver` and the `sender`.
- In context-oriented programming [43], it is as for subjective programming, but also taking into account one ore more context dimensions.

The same approach can be taken to study the design patterns often referred to in object-oriented programming [40]. For example, the PROTOTYPE Object Creational Pattern creates a new object by cloning an existing object. There are two fixed dimensions: `subclass` and `prototype`, but one can add as many context dimensions as one wishes to get a slightly different clone of the prototype.

Similarly, the Structural and Behavioral Patterns described in [40] essentially define how a message can be translated by a receiving object of one class into another message to be passed on to another object of another class. For each pattern, there is a restricted set of fixed dimensions, and possibly an additional unspecified set of further dimensions for refinement purposes.

As for concurrency, studying it from an indexical perspective will require experimenting with multiple timed systems, which was discussed in Chapter 6.

Conclusions

At electric speeds of data processing, we become aware of environments for the first time. We call them "parameters". It all began in electronic research when it was discovered that the instruments of observation distorted the data. Now we know that any environment acts like the instrument of observation.

Marshall McLuhan, 1964 [61].

The multidimensionality implicit in Cartesian programming has been shown to be remarkably versatile. Given any problem, there is a small, fixed set of dimensions that define the problem, with an arbitrarily large additional set of dimensions that parameterize that problem.

The set of topics that follows on from the current document is very large. Instead of trying to cover all possible issues, we examine in some detail two issues that have not been discussed substantially in the thesis: 1) data structures and types, and 2) algorithms and complexity.

Data structures and types

The current TransLucid manipulates three kinds of objects: atomic objects, i.e., $(type, value)$ pairs; tuples, i.e., mappings from values to values; and hyperdatons. The first two are completely constructed as expressions are evaluated, while the hyperdatons, typically being infinite, are only sampled as needed. Type checking is done dynamically, to ensure that operators are applied to objects of the right type, and there is no static type inference.

With respect to type inference, we could, of course, have added strong static typing to the language, both for variables and for dimensions, and previous interpreters have had this. However, given that hyperdatons are infinite, multidimensional entities, it is not always clear that every cell in a hyperdaton should have the same type. This is clearly not the case for commercial spreadsheets, for example. The proper solution requires the declaration of types of variables to be context-dependent, as is the declaration of definitions for variables; this has already been done experimentally in one interpreter. Type inference would in general then not be decidable, and so typing would require some interesting mix of static and dynamic typing. Problems requiring very stringent running constraints would have to be typed fully statically, while more experimental programs could run with dynamic typing.

An interesting aspect of the bestfitting process within the current interpreter is that it is also responsible for choosing which specific host language function should be called when an operator appears in the source program. This process considers a type to be not a name but, rather, a *set of values*, and that a value may actually belong to multiple different sets. Furthermore, the bestfitting process implicitly assumes that a single object can be understood as belonging to multiple "classes". The theory that is most promising is that introduced by Adi Shamir and William Wadge [85], but details of adapting our current implementation to follow this path are not fully worked out.

Inductively defined abstract data types, as used in languages such as Haskell and OCaml, can be implemented in TransLucid using tuples. Different types would be distinguished using, for example, the `type` dimension, while the different constructors for a given type would be distinguished by, say, the `constructor` dimension. Syntactic sugar would allow these to be parsed and printed in ways familiar to most functional programmers.

Content as index

The interaction between "atomic values" and tuples can be very interesting, even for subjects that are considered to be completely solved. We examine here the question of "characters" in documents. Consider, for example, character 'O', typically encoded in a document as a Unicode character. It will appear in a byte stream as the UTF-8 byte `4F`, corresponding to the tuple:

```
[
  Type: UnicodeChar
  CodePoint: U+004F
]
```

This information can then be used to lookup this character in the Unicode Data database [19], where we will find the following entry:

```
004F;LATIN CAPITAL LETTER O;Lu;0;L;;;;;N;;;;006F;
```

If we ignore the default values, and use the explanations given in [20], the above line corresponds to the tuple:

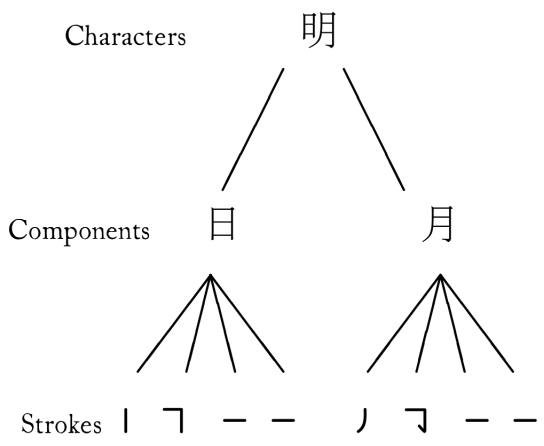
```
[
  CodePoint : U+004F
  Name : "LATIN CAPITAL LETTER O"
  GeneralCategory : "Lu"
  CanonicalCombiningClass : 0
  BidiClass : "L"
  BidiMirrored : "N"
  SimpleLowecaseMapping : U+006F
]
```

i.e., the official Unicode Standard name of 'O' is 'LATIN CAPITAL LETTER O'; it is an uppercase letter; it is not reordered for combining purposes; it is a strong left-to-right character and not mirrored for bidirectional text; and its lowercase mapping is code point `U+006F` (the letter 'o').

With this approach, the set of characters can grow as needed, and is not limited to Unicode or some other established character set. After all, Unicode places restrictions not only on the set of possible dimensions, but also on the values to be taken by these dimensions.

For example, it is not possible to encode all of the Chinese characters, for one can always create new ones, and there are many cases of *hapax legomena*, i.e., characters only appearing once in the historical corpus which are not yet encoded. Our model allows the use of a character description language to register unencoded characters [97].

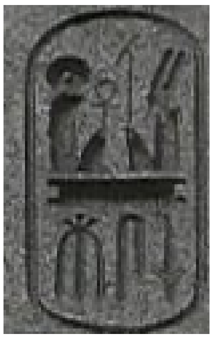
For example, suppose that the character for '*míng*', meaning *bright* or *clear*, were not encoded. Then we could break it down into its two radicals, '*rì*' (*sun*) and '*yuè*' (*moon*), as is shown below:



The full set of strokes needed for ‘*míng*’ is given below (Note: just one possible representation):

```
[
  Type: ChineseGlyph
  RadicalOrder: LeftRight
  Left:  [
           0: "shù"
           1: "héng zhé"
           2: "héng"
           3: "héng"
         ]
  Right: [
           0: "piĕ"
           1: "héng zhé gōu"
           2: "héng"
           3: "héng"
         ]
]
```

Sometimes, the abstraction of ‘character’ is not sufficient, nor even an abstract visual description. This is the case, for example, when studying steles in archæology: exact images are required for presentation, discussion and analysis, yet these still need to be treated as ‘characters’. The photograph below presents one such example, a cartouche at the bottom of the obelisk in the center of Paris’s Place de la Concorde [17], originally from Luxor Temple in Egypt.



The following description suffices:

```
[
  Type: EgyptianCartouche
  CurrentLocation: "Place de la Concorde, Paris"
  CurrentDate: 1833
  OriginalLocation: "Temple, Luxor"
  OriginalDate: "3300BP"
  PNGfile: "EgyptianCartouche.png"
]
```

The previous examples should demonstrate that this approach is extensible as needed, in every direction. Should further dimensions be needed, they can be added at any time. What is considered to be an atomic value is simply a convention, and a ‘magnifying glass’ can be used for further detail.

The context is the link

When we think of a programming environment, we do not just think about the language, but also the libraries that come with that language. For example, the success of Smalltalk came in no small part from the containers that were provided in the distribution. Similarly, the success of `C++` cannot be disassociated from the Standard Template Library (STL) and its generic algorithms.

For TransLucid, we are currently developing the *indexed container*, in which every cell in the container has a unique index allowing direct access to it and which is robust to editing, in the sense that future modifications to the container will not affect existing indexes.

The STL provides a number of containers, including vectors, lists, deques, and so on. We consider all of these to be *linear* containers, in the sense that we can order all of the elements in a straight line, and that there is a beginning and an end to this order. However, the insertion properties of the different containers are very different: some only allow insertion at the end or the beginning, others anywhere.

In our vision, all of these are special cases of the one-dimensional indexed container, described as follows:

- A container uses the set $\mathbb{Q}$ of rationals as the set of possible dimensions. In fact, only those rationals with powers of 2 as denominator are needed.
- Every container responds to the dimensions -1 and 1, which respectively represent before the container and after the container, neither ever pointing to an actual value.
- The first element is added at dimension 0.
- If the minimum element is at dimension q, then an insertion before q is placed at $(q-1)/2$.
- If the maximum element is at dimension q, then an insertion after q is placed at $(q+1)/2$.
- If an element is to be placed between adjacent elements p and q, it is placed at $(p+q)/2$.
- If we keep track of the evolution of time as this structure is created, every element within the container has a unique address which is robust.

This approach can be extended to higher dimensions. For example, using a second-dimension would allow the creation of trees and dags in which one could insert nodes along a path between existing nodes. In this case, the initial seed would be placed at position $(0,0)$. Normal computer-science trees would continue to grow down, but we could also extend these to allow growing up, much as do real trees, but also back down again, as with the banyans found in tropical regions of the world.

The robustness requirement is crucial for the development of Web-enabled applications. There is an increasing trend towards the use of permanent URLs for documents, as in the use of Digital Object Identifiers (DOIs) in the publishing industry. Using the mechanism above would allow the concept of permanent URL to become even more precise, as it could accurately point to individual characters within documents; Ted Nelson would be proud.

Algorithms and complexity

In traditional complexity theory, a problem is defined in terms of its "size" n, and its complexity is defined in terms of a function in n: for example, quicksort is $\mathcal{O}(n^2)$ worst-case and $\mathcal{O}(n \log n)$ average-case time complexity. In *parametric* complexity theory, introduced by Michael Fellows and Rodney Downey [29], a problem is defined in terms of its size n and a parameter k. A parameterized problem $L \subset \Sigma^* \times \Sigma^*$ is *fixed-parameter tractable* if it can be determined in $f(k) \cdot n^{\mathcal{O}(1)}$ time whether or not $(x, k) \in L$, where f is a computable function only depending on k. A good introduction to the theory is the book *Invitation to Fixed-Parameter Algorithms*, by Rolf Niedermeier [64].

There are many problems that are known to be NP-complete which turn out to be fixed-parameter tractable, and for which realistic applications of the problem ensure that the parameter k is sufficiently small that the problem becomes effectively tractable on real data.

The intuition is that the hard part of these problems is in the parameter k part, not in the overall size n of the problem. As a result, a common practice in fixed-parameter algorithm design is a process called *kernelization*, in which the problem is concentrated by cutting out the easier bits, leaving the separated hard parts, in which exhaustive search is not uncommon. This kernelization is a form of *preprocessing*, in which the problem is "massaged" so that it is more amenable to the main algorithm.

In Part II of his book (the main part), Niedermeier goes to great lengths to show how different approaches can be taken to creating fixed-parameter algorithms for a wide range of problems. One of the important points that he makes is that preprocessing is typically not something that is run just once, but rather the main algorithm and preprocessing must be interleaved. At each step, to reduce the search space of what remains to be solved, one should clear out the easy parts, leaving what is still known to be hard.

In recent discussions, Fellows proposes to develop a catalog of *Total Victory Maps*, in which the parameterized complexity approach succeeds in developing the best worst-case time bounds, for all inputs [39]. In the discussion, he raises the question, "How many dimensions are relevant to such mappings?"

This is where Cartesian programming may be relevant, using a process that we call *iterative bestfitting*, which we now describe. Suppose that there is a problem to solve, the input is the data D_0 and that there is a family of algorithms $\mathcal{A}$ ($\ni A$). Then the resolution of this problem is undertaken iteratively. At each step i:

- Apply to D_i a preprocessing step $B_i(D_i)$, and in so doing, calculate a set of relevant parameters $p_{i1}, \ldots, p_{in_i}$.
- Use bestfitting over the $p_{i1}, \ldots, p_{in_i}$ to find algorithm A_i. Run $A_i(D_i)$, returning D_{i+1}.
- If there are several possible best algorithms $A_{i1}, \ldots, A_{im_i}$, then they can all be run in parallel, and then D_{i+1} would be the either the first or the best answer returned by $A_{ij}(D_i)$.
- If D_{i+1} is a solution, or it is clear that there is no solution, halt. Otherwise, start step $i + 1$.

The iterative bestfitting approach is well-suited to developing algorithms in which it is difficult to determine statically which approach will be best in which conditions. The approach is well-suited both for theoretical complexity work as well as for the development of real algorithms for real problems, since at each iteration, the bestfitting may take into account not just the available algorithms, but the knowledge of the parameters computed by the preprocessing step, as well as the results from previous iterations.

From the analysis point of view, the fact that bestfitting of the algorithm takes place at each iteration does not facilitate anything, as it makes any recurrence equation harder to solve. Nevertheless, we consider that the iterated bestfit approach would facilitate the development of Total Victory Maps. Moreover, we can envisage developing this idea even further by taking into account variants of problems in the mapping process: this just means adding more dimensions.

Whither from here?

Apart from the topics of data and complexity, there are many other topics which we have not considered in this thesis.

For example, there is little discussion of static analysis. This very important topic can be applied in numerous manners to the development of TransLucid, including type checking, space and time optimization and formal verification. Given that TransLucid is a completely declarative language, and that variables already have multiple declarations, we can add further declarations. In this case, during the bestfitting process, should multiple declarations be "best", then all of them would be applied, and any evaluation or verification would have to ensure that all of the "best" declarations yield consistent results in a given context.

Similarly, there is no discussion of what we call "search-based programming" (unfortunately the term is consecrated elsewhere), which includes any form of programming in which a considerable part consists of searching through large amounts of data, as in logic programming, database programming and data mining. Currently, we have no semantics for such forms of programming, although the use of multiple dimensions is clearly relevant to all of them.

In general, the implementation of a given problem can itself be understood as requiring the parameterization of the solution space, based on the available components for processing, in the memory hierarchy and for communication. Research in this area should yield fruitful results, considering the wide variety of multiprocessors being developed for the current market place.

When Descartes introduced his coordinate sytem, it greatly simplified the presentation and resolution of a number of existing problems in geometry, and allowed the definition of much more difficult problems that previously could not be expressed. So far, each part of computer science that we have examined through the Cartesian lens has allowed us new insights. Will Cartesian programming also lead to the future specification of problems hitherto unexpressible?

Bibliography

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, 2006.

[2] E. A. Ashcroft and W. W. Wadge. Lucid—A formal system for writing and proving programs. *SIAM Journal on Computing*, 5(3):336–354, September 1976.

[3] E. A. Ashcroft and W. W. Wadge. Lucid, A Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[4] Edward A. Ashcroft. The Lucid Language: Past, Present, and Future. In *Proceedings of the 5th ISLIP*, pages 1–15, San Francisco, USA, Apr 1992. IEEE Computer Society. `islip.web.cse.unsw.edu.au/1992`.

[5] Edward A. Ashcroft, Anthony A. Faustini, Rangaswamy Jagannathan, and William W. Wadge. *Multidimensional Programming.* Oxford University Press, New York, 1995.

[6] Edward A. Ashcroft and William W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[7] Varunan Balasingham, Gabriel Ditu, and Simon Hudson. Intensionality and the object-oriented paradigm. Honours Thesis, Computer Science, UNSW, 2001.

[8] Jarryd Beck. Cartesian programming: From theory to implementation. Honours Thesis, Computer Science, UNSW, 2010.

[9] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[10] I. M. Bocheński. *Ancient Formal Logic.* North-Holland, Amsterdam, 1951.

[11] I. M. Bocheński. *A History of Formal Logic.* University of Notre Dame Press, Notre Dame, Indiana, 1961. Translated from the German *Formale Logik.*

[12] Gordon D. Brown. Intensional HTML 2: A practical approach. Master's thesis, Department of Computer Science, University of Victoria, Canada, 1998.

[13] Vannevar Bush. As we may think. *Atlantic Monthly*, July 1945.

[14] Rudolph Carnap. *Meaning and Necessity.* University of Chicago Press, 1956. Seventh Impression 1975.

[15] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. A declarative language for programming synchronous systems. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages, POPL 1987*, pages 178–188, Munich, January 1987.

[16] Aggelos Charalambidis, Athanasios Grivas, Nikolaos Papaspyrou, and Panos Rondogiannis. Efficient intensional implementation for lazy functional languages. *Mathematics in Computer Science*, 2(1):123–141, 2008.

[17] Wikipedia Commons. Image of Obelisk in Paris, 2006. `upload.wikimedia.org/wikipedia/commons/3/31/Obeliskparis.jpg`.

[18] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[19] Unicode Consortium. `www.unicode.org/Public/UNIDATA/UnicodeData.txt`, 2009.

[20] Unicode Consortium. `www.unicode.org/reports/tr44/#UnicodeData.txt`, 2009.

[21] N. J. Cutland. *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press, 1980.

[22] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

[23] Ole-Johan Dahl, Edsger Wybe Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, London, 1972. ISBN 0122005503.

[24] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms.* McGraw Hill, 2007.

[25] Jean Dhombres. La banalidad del referencial cartesiano. In Carlos Álvarez J. and J. Rafael Martinéz Enríquez, editors, *Descartes y la ciencia del siglo XVII.* siglo veintiuno editores, México, 2000.

[26] Edsger Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[27] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976. ISBN 013215871X.

[28] Gabriel Ditu. *The Programming Language TransLucid.* PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, March 2007.

[29] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Springer, 1999.

[30] David R. Dowty, Robert E. Wall, and Stanley Peters. *Introduction to Montague Semantics.* D. Reidel, Dordrecht, Holland, 1981.

[31] W. Du and W. W. Wadge. A 3D spreadsheet based on intensional logic. *IEEE Software*, 7(3):78–89, 1990.

[32] W. Du and W. W. Wadge. The eductive implementation of a three-dimensional spreadsheet. *Software–Practice and Experience*, 20(11):1097–1114, 1990.

[33] Doug Engelbart. Personal communication, 2004.

[34] Esterel Technologies. Home page. `www.esterel-technologies.com`. Last visited 8 December 2007.

[35] A. A. Faustini and W. W. Wadge. Intensional programming. In J. C. Boudreaux, B. W. Hamil, and R. Jenigan, editors, *The Role of Languages in Problem Solving 2.* Elsevier North-Holland, 1987.

[36] Anthony A. Faustini and Rangaswamy Jagannathan. Indexical Lucid. In *Proceedings of the 4th ISLIP*, pages 19–34, Menlo Park, USA, Apr 1991. SRI International. `islip.web.cse.unsw.edu.au/1991`.

[37] Anthony A. Faustini and Rangaswamy Jagannathan. Multidimensional Problem Solving in Lucid. Technical Report SRI-CSL-93-03, Computer Science, SRI International, Menlo Park, USA, 1993.

[38] Antony A. Faustini and William W. Wadge. An eductive interpreter for lucid. In *PLDI*, pages 86–91. ACM, 1987.

[39] Michael Fellows. New ideas column. *Parameterized Complexity News*, pages 6–7, September 2009. `mrfellows.net/Newsletters/09_Sept_News.pdf`.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[41] Manolis Gergatsoulis and Panos Rondogiannis, editors. *Intensional Programming II*, volume II. World Scientific, Singapore, 2000. Based on the papers at ISLIP'99.

[42] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[43] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[44] Lapyu Kenneth Ho, Ian Su, and Patrick Leung. Multidimensional Linux library system. Honours Thesis, Computer Science, UNSW, 2001.

[45] *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland.* IEEE Computer Society, 2008.

[46] Rangaswamy Jagannathan and Chris Dodd. GLU Programmer's Guide. Technical report, Computer Science, SRI International, Menlo Park, USA, 1996. Version 1.0.

[47] Rangaswamy Jagannathan, Chris Dodd, and Iskender Agi. GLU: A High-Level System for Granular Data-Parallel Programming. *Concurrency: Practice and Experience*, 9(1):63–83, Jan 1997.

[48] Gilles Kahn. The semantics of simple languages for parallel programming. In *Information Processing 74*, pages 471–475. North-Holland, 1974.

[49] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998. North-Holland, 1977.

[50] Saul A. Kripke. *Naming and Necessity.* Harvard University Press, Cambridge, Mass., 1980.

[51] Peter Kropf, Gilbert Babin, John Plaice, and Herwig Unger, editors. *Distributed Communities on the Web*, volume 1830 of *Lecture Notes in Computer Science.* Springer, Berlin, 2000. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[52] P.G. Kropf and J. Plaice. WOS communities—Interactions and relations between entities in distributed systems. In *Distributed Computing on the Web, DCW'99*, pages 163–167, Rostock, Germany, June 1999.

[53] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[54] m. c. schraefel. *Talking to Antigone.* PhD thesis, University of Victoria, Victoria, Canada, 1997.

[55] Blanca Mancilla. A new approach to on-line mapping. In *Proceeding of IWHIT'00, Seventh International Workshop on Human Interface Technology 2000*, pages 19–23. University of Aizu, Aizu-Wakamatsu, Japan, November 2000.

[56] Blanca Mancilla. A new approach to interactive mapping. In Michitake Hirose, editor, *Human-Computer Interaction — INTERACT'01*, pages 644–647. IOS press for IFIP, 2001.

[57] Blanca Mancilla. Omega for multilingual mapping. In Bolek and Przechlewski, editors, *Proceedings of XIII European TEX Conference April 29–May 3, 2002, Bachotek, Poland*, pages 60–63. GUST, 2002.

[58] Blanca Mancilla. *Intensional Infrastructure for Collaborative Mapping.* PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2004.

[59] Blanca Mancilla and John Plaice. Possible worlds versioning. *Mathematics in Computer Science*, 2(1):63–83, 2008.

[60] m.c. schraefel, Blanca Mancilla, and John Plaice. Intensional hypertext. In Gergatsoulis and Rondogiannis [41], pages 40–54. Based on the papers at ISLIP'99.

[61] Eric McLuhan and Frank Zingrone, editors. *Essential McLuhan.* House of Anansi Press, Toronto, 1995.

[62] Gordon E. Moore. Excerpts from A Conversation with Gordon Moore: Moore's Law. Video Transcript, Intel, 2005.

[63] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[64] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006.

[65] Calvin B. Ostrum. *The Luthid 1.0 Manual.* Department of Computer Science, University of Waterloo, Ontario, Canada, 1981.

[66] Joey Paquet. *Intensional Scientific Programming.* PhD thesis, Department of Computer Science, Laval University, Québec, Canada, April 1999.

[67] Joey Paquet and Peter G. Kropf. The GIPSY architecture. In Kropf et al. [51], pages 144–153. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[68] Joey Paquet and John Plaice. *The semantics of dimensions as values*, pages 259–273. Volume II of Gergatsoulis and Rondogiannis [41], 2000. Based on the papers at ISLIP'99.

[69] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.

[70] Anthony L. Peratt. *Physics of the Plasma Universe.* Springer, New York, 1992.

[71] John Plaice. Multidimensional Lucid. In Kropf et al. [51], pages 154–160. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[72] John Plaice and Peter Kropf. *Intensional communities*, pages 292–295. Volume II of Gergatsoulis and Rondogiannis [41], 2000. Based on the papers at ISLIP'99.

[73] John Plaice and Blanca Mancilla. Collaborative intensional hypertext. In *Hypertext*, pages 91–92. ACM, 2004.

[74] John Plaice, Blanca Mancilla, and Gabriel Ditu. From Lucid to TransLucid: Iteration, dataflow, intensional and Cartesian programming. *Mathematics in Computer Science*, 2(1):37–61, 2008.

[75] John Plaice, Blanca Mancilla, Gabriel Ditu, and William W. Wadge. Sequential demand-driven evaluation of Eager TransLucid. In IEEE Computer Society [45], pages 1266–1271.

[76] John Plaice, Paul Swoboda, and Ammar Alammar. Building intensional communities using shared context. In Kropf et al. [51], pages 55–64. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[77] John Plaice and William W. Wadge. A new approach to version control. *IEEE Transactions on Software Engineering*, 19(3):268–276, March 1993.

[78] John Plaice and William W. Wadge. A UNIX tool for managing reusable software components. *Software—Practice and Experience*, 23(9):933–948, September 1993.

[79] Toby Rahilly. Experiments in concurrent implementations of TransLucid. Honours Thesis, Computer Science, UNSW, 2007.

[80] Toby Rahilly and John Plaice. A multithreaded implementation for TransLucid. In IEEE Computer Society [45], pages 1272–1277.

[81] Panos Rondogiannis and William W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, January 1997.

[82] Panos Rondogiannis and William W. Wadge. Higher-order functional languages and intensional logic. *Journal of Functional Programming*, 9(5):527–564, May 1999.

[83] Dana Scott. Advice on modal logic. In Karel Lambert, editor, *Philosophical Problems in Logic*, pages 143–173. D. Reidel Publishing Company, Dordrecht, Holland, 1970.

[84] Victor Selivanov. Wadge reducibility and infinite computations. *Mathematics in Computer Science*, 2(1):5–36, 2008.

[85] Adi Shamir and William W. Wadge. Data types as objects. In Arto Salomaa and Magnus Steinby, editors, *ICALP*, volume 52 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 1977.

[86] Randall B. Smith and David Ungar. A simple and unifying approach to subjective objects. *TAPOS*, 2(3):161–178, 1996.

[87] Yannis Stavrakas, Manolis Gergatsoulis, and Panos Rondogiannis. Multidimensional XML. In Kropf et al. [51], pages 100–109. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[88] Paul Swoboda. Practical languages for intensional programming. Master's thesis, Department of Computer Science, University of Victoria, Canada, 1999.

[89] Paul Swoboda. *A formalization and implementation of distributed Intensional Programming.* PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2003.

[90] Senhua Tao. TLucid and Intensional Attribute Grammars. In *Proceedings of the 6th ISLIP*, pages 94–106, Laval University, Québec, Canada, Apr 1993. `islip.web.cse.unsw.edu.au/1993/`.

[91] Richmond H. Thomason, editor. *Formal Philosophy: Selected Papers of Richard Montague.* Yale University Press, 1974.

[92] William W. Wadge. The Origins of Lucid. Presentation to the Institut für Informatik, Christian-Albrechts-Universität zu Kiel, Germany, 8 February 2007.

[93] William W. Wadge. *Intensional logic in context*, pages 1–13. Volume II of Gergatsoulis and Rondogiannis [41], 2000. Based on the papers at ISLIP'99.

[94] William W. Wadge. Intensional Markup Language. In Kropf et al. [51], pages 82–89. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[95] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[96] William W. Wadge, Gordon D. Brown, m.c. schraefel, and Taner Yıldırım. Intensional HTML. In E.V. Munson, C. Nicholas, and D. Wood, editors, *Principles of Digital Document Processing*, volume 1481 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[97] Wai Wong, Candy L. K. Yiu, and Kelvin C. F. Ng. Typesetting rare Chinese characters in LaTeX. *TUGboat*, 24(3):582–587, 2003. `www.tug.org/TUGboat/Articles/tb24-3/wong.pdf`.

[98] Taner Yıldırım. Intensional HTML. Master's thesis, Department of Computer Science, University of Victoria, Canada, 1997.