ON THE DESIGN OF FINITE-STATE TYPE SYSTEMS

by

ALEXANDER IAN SMITH

A thesis submitted to the University of Birmingham for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
July 2015

Abstract

Practical computers have only finite amounts of memory. However, the programs that run on them are often written in languages that effectively assume (via providing constructs such as general recursion) that infinite memory is available, meaning that an implementation of those programs is necessarily an approximation.

The main focus of this thesis is on the use of contraction: the ability to use a function parameter more than once in the body of that function (or more generally, to mention a free variable more than once in a term). Unrestricted contraction is a common reason for a language to require unbounded amounts of memory to implement.

This thesis looks at a range of type systems, both existing and new, that restrict the use of contraction so that they can be implemented with finite amounts of state, identifying common themes, and explaining and suggesting solutions for common deficiencies. In particular, different restrictions on contraction are seen to correspond to different features of the language's implementation.

Contents

List of Figures

Notation

θ, θ'	types (sometimes τ is used for base types when the base types are tags)
x, y, z	variables, in general
M, N	terms
Γ, Δ	contexts (lists of free variables and their types)
$\Gamma \vdash M : \theta$	type judgements: "M has type θ if the free variables are as in Γ"
∇, ∇'	type derivations
$P \triangleq Q$	"P is defined to mean Q"
j, k, l	integers or nonnegative integers; or semiring elements more generally
J, K	sets of integers or nonnegative integers, or semirings more generally
τ, τ'; υ, υ'	tags (for type systems that use tags); tag components
$\mathbb{N}$	the set of nonnegative integers ($\mathbb{N} \triangleq \{j \in \mathbb{Z} \mid j \geq 0\}$)
$\#$	freshness/non-overlapping (of free variables, time intervals, etc.)
$\otimes$; $\Rightarrow$	tensor (both in a type system, and in a category); adjoint of the tensor
$\rightarrow$; $\multimap$; $\times$	function formation; linear/affine function formation; product formation
$\langle M, N \rangle$	product formation on terms (or more generally on elements of sets)
π_1; π_2	left and right projection (extraction of components from products)
$M[x/y]$	"M, except all uses of y are replaced by x"
$\oplus$; $\odot$; $\mathbf{0}$; $\mathbf{1}$	semiring addition; multiplication; additive unit; multiplicative unit
$\cap$; $\cup$; $\uplus$; $\setminus$	intersection; union; disjoint union; set difference ($j \in J \setminus K$ iff $j \in J \wedge j \notin K$)
$\bullet$	arithmetic operations
$[\![M]\!]$, $[\![\theta]\!]$	denotations of terms, and of types
$\emptyset$; $[]$; $\{\}$; ε	empty set; empty multiset; empty function (domain $\emptyset$); empty sequence
$::$	sequence concatenation
$\mathscr{C}$, $\mathscr{D}$, $\mathscr{E}$	categories
A, B, C, D	objects of categories
f, g, h	morphisms of categories, or functions, depending on context
F, G	functors
α, β, γ; α^{-1}	transformations; inverse of a natural isomorphism
$f : A \rightarrow B$	domain and codomain: "f is a function/functor/transformation from A to B"
$\alpha : A \cong B$	"α is an isomorphism from A to B"
$f : A \longrightarrow B$	"f is a morphism from A to B"
$\prec$, $\prec'$	relations, in general
$\prec^*$	transitive closure of $\prec$ (i.e. $\tau \prec^* \tau'$ iff $\tau \prec \tau' \vee (\tau \prec^* \tau'' \wedge \tau'' \prec^* \tau')$)

Statement of Contributions

While working on this thesis, I published the following papers:

- *Geometry of Synthesis II: From Games to Delay-Insensitive Circuits* ([19], joint work with D. R. Ghica). This paper describes a synthesis technique that allows programs written in Basic Syntactic Control of Interference (bSCI) to be compiled into delay-insensitive asynchronous hardware. Most of the paper was written by Ghica; my main contribution was the statement and proof of Theorem 5.3 of that paper (which produces both the main correctness results of that paper as corollaries). I did not include the content of this paper in this thesis.

- *Geometry of Synthesis III: Resource Management through Type Inference* ([20], joint work with D. R. Ghica). This paper describes techniques for translating a program from Idealized Concurrent Algol (ICA) that happens to type in Syntactic Control of Concurrency (SCC) to a slightly extended version of bSCI called "SCC(1)". It contains two main results: a type inference algorithm for SCC in its chapter 3 (an algorithm which I discovered; some of it is inspired by standard type inference techniques, but the way it solves the constraints is new); and a serialization algorithm from SCC to SCC(1), which is mostly due to Ghica (although I produced the transformation used to handle subtyping in this step). In this thesis, I adapt the type inference algorithm to a different type system ("Bounded ICA") in order to clarify the presentation, and it can be found in Section 6.2. The subtyping algorithm can be found in Section 6.5 (specifically Figure 6.5.3), again adapted to the Bounded ICA setting. I also produced a counterexample for this paper, which is given here as Fact 6.2.13.

- *Geometry of Synthesis IV: Compiling Affine Recursion into Static Hardware* ([22], joint work with D. R. Ghica and S. Singh). This paper describes two different techniques for implementing recursion in hardware. My contribution was the basic idea of the construction in chapter 5 of that paper, which implements affine recursion for non-concurrent sections of a program via indexing all stateful circuits using a counter; some of the details were worked out by Ghica. In this thesis, I discuss the construction in question in Section 9.2.

- *Bounded Linear Types in a Resource Semiring* ([21], joint work with D. R. Ghica). This paper presents an extension of the call-by-name fragment of Bounded Linear Logic to the corresponding fragment of a new type system, Semiring-bounded Linear Logic (SBLL). My main contributions were the definition of the type system itself, and the instance of it that uses contractive affine transformations to describe timing; Ghica's main contribution was the study of the categorical semantics. Both authors contributed a large number of minor details. In this thesis, SBLL is discussed in Section 7.1. It should also be noted that another research group (A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic) discovered SBLL independently at about the same time; their work on the subject is published in [4] (which was published in the same issue of the same journal).

This thesis also contains many previously unpublished new contributions:

- The proof from [20] that the type inference algorithm for SCC/Bounded ICA is sound (i.e. never produces an invalid type for a term) is extended with a proof that the algorithm is also complete (i.e. always finds a type for a term, if one exists), thus showing that type inference for Bounded ICA is decidable (Corollary 6.2.11).

- The observation that when designing a type system that describes some detail of terms in another type system, then a term that types without polymorphism in the original type system might need polymorphism or a similar construct to type in the descriptive type system. This idea is formalized as Theorem 6.3.2, which proves that a type system cannot be more descriptive than simply typed lambda calculus and yet type all the same terms if its Application and Contraction rules obey some reasonable conditions. (This in turn explains why most finite-state type systems have terms which they would expect to be able to type, but cannot).

- The introduction of new type systems that aim to work around the issues with polymorphism explained in the previous paragraph (and in some cases, other issues as well) via the use of bounded intersection typing; Intersecting ICA (Section 6.4) as a variant of Bounded ICA, and likewise Intersection/Semigroup-bounded Linear Logic (ISBLL, Section 7.4) as a variant of SBLL, and Tagged Control of Interference (TCI, Section 8.3) as a variant of SCI.

- A description of how and why SBLL needs to be modified in order to support pipelining behaviour, in Section 7.2.

- The observation in Section 7.3 that when exponential-like operators ("tags", in this thesis) are combined with tensors, there are important reasons to be able to put the tag both inside and outside the tensor, thus producing an explanation of why many descriptive finite-state type systems have problems handling tensors; together with a suggested solution, via the use of a larger fragment of SBLL than merely the call-by-name fragment.

- An explanation in Section 8.2 of how bSCI is effectively encoding existential types using its products, thus explaining why there is no obvious way to map it onto a framework like SBLL or ISBLL without changes that go beyond merely selecting a semiring or semigroup to parameterize the type system.

- A description of the mathematical problems in separate compilation with SCI-like languages, and a proposed solution, in Section 10.3.

- And finally, the conclusion (in Section 11.1) that in finite-state type systems, language features tend to be expressed as particular forms of contraction rules.

Part I

INTRODUCTION

Chapter 1

INTRODUCTION

1.1 Structure of this thesis

There are four main parts to this thesis. Part I contains the motivation for the thesis, and discusses some other approaches that have been tried for producing a similar result that do not fit with the main flow of the thesis. Part II contains the mathematical background; this is a statement of many existing results that predate the thesis and that will be relevant, partly for the benefit of readers who do not know them, and partly to clarify notation. Part IV talks about the issues in trying to apply the theoretical results in this thesis in practice, and also contains conclusions.

The main body of the thesis, including the vast majority of the new contributions, is in Part III. This section discusses a number of mathematical type systems, sorted according to what sort of contraction they allow (which will be seen to reflect what sort of language features they contain). These type systems are nearly all variants of Idealized Concurrent Algol (abbreviated to ICA in this thesis to save space), a pre-existing type system discussed in Subsection 3.3.2. All the type systems are effectively subsets of (full) ICA, which has no restrictions on contraction, and supersets of Affine ICA, an ICA variant that has no contraction. Thus, the other type systems all support contraction to a limited extent.

Chapter 5 discusses Affine ICA, which admits no contraction; being the simplest of the type systems, this acts as a sort of base from which to look at the others. Chapter 6 considers bounded contraction, in which contraction is limited via requiring a finite number of copies; Chapter 7 considers synchronized contraction, in which two terms can be contracted if they can be allocated different time intervals in which to run; and Chapter 8 considers sequential faster method. This opened up a new field of mathematics, computer science, which built on the field of mathematical logic to study the programs that computers can execute.

In mathematics, there is typically no particular reason to place any sort of bound on how long an algorithm might take to execute, or how much storage space might be required to record all the information that it needs to refer to; the only requirement on an algorithm is typically that it eventually terminates. Nowadays, methods of adapting an algorithm or computation to run on a general-purpose computer usually make the same assumptions; if a computer program needs to store some information, it just stores it, and leaves the details up to the computer's operating system or the programming language implementation. Various implementation techniques for programming languages have been developed in order to automate the process of data storage.

However, this idealization of a computer can break down in practice. Even on general-purpose computer hardware, a very large calculation, or a program containing a "memory leak" (in which it falsely claims that information must be stored because it might be needed in the future), can lead to failure due to memory exhaustion. The situation becomes much worse when generating hardware for a specific task; one of the major reasons to use something other than a general-purpose computer is to avoid the bottlenecks that result from ensuring that all calculations have access to any part of memory at any given time, which means that special-purpose hardware often needs an exact accounting of how much storage is needed for each part of the program.

An alternative approach is to choose, for our mathematical models, programming languages that place limits on which programs are allowed, by ensuring that the programs can be implemented on a finitely large computer, without having to hope that they fit in to the amount of memory available. (This thesis calls these languages "finite-state languages", by analogy with finite-state machines.) At the time I started working on my PhD, several such models were already available; thus, I assumed that this was a solved problem, and started building on it, in order to try to produce a hardware-based implementation of a particular programming language. (My experience doing that is presented as a case study in Chapter 10.) The motivating scenario for this thesis, then, is that of "what programming languages enable all programs written in them to be implementable in hardware?". However, the results should be more generally applicable.

However, I ran into problems trying to use the existing languages as part of my hardware compilation project. One of these problems was that different languages are susceptible to different implementation techniques, and many such techniques are very inefficient. For example, in Chapter 6, we look at the technique of making multiple copies of a subprogram and using each copy only once, which will in most cases not be the most efficient way to implement a program. Alternatively, it is possible to use a language that can be implemented efficiently, but at the risk of being too restrictive to write useful programs (Affine ICA, discussed in Chapter 5, is a good example of this). The solution is to produce a range of languages, with their own corresponding implementation techniques, so as to be able to choose our trade-off between power and efficiency based on the needs of the programmer.

The other problem I ran into was that some of the existing languages were unsuitable for the purpose of using them as a finite-state language. For example, I based my project to implement programs in hardware on Syntactic Control of Concurrency (SCC, introduced by Ghica, Murawski and Ong in[18]); but SCC turned out to disallow some programs that could be implemented with techniques that it seemed like it should allow. Although this had been noted even in the paper introducing the language, I initially assumed that it was just an anomaly. However, this problem also turned up in several of the other languages I created. The problem is actually quite fundamental, and not just a quirk of SCC; it turns out that it is due to an incompatibility between some assumptions that are commonly made in type system design and the goal of languages like SCC. (This result is presented in Section 6.3, and is one of the major results of this thesis.) This was not the only problem I identified that was common to many of the type systems in the area, either.

Thus, this thesis aims to look at finite-state languages – specifically, their type systems – and identify common themes and problems. It considers both many existing such languages, and languages created for the purposes of this thesis. In particular, a common pattern emerges, in which features in the implementation of a language are reflected in the restrictions on the Contraction rules of those language's type systems. Common pitfalls are also identified, explaining what problems can come up in the design of finite-state type systems, why they happen, and how to correct them.

1.3 Alternative approaches

One of the goals of this thesis is to look at and discuss common features in finite-state languages. This means that for most of the relevant papers, it makes more sense to discuss them in detail as their contents become relevant; we discuss many of the relevant existing papers ([40, 42, 17]) later on in Chapter 3, and even more are considered either briefly or in depth in the main body of the thesis ([14, 23, 28, 35, 34, 40, 41]). This section briefly discusses some other approaches for accomplishing similar goals to those in this thesis, but via different methods, and thus which will not be relevant later on.

One of the more obviously finite-state constructs is a finite-state machine; these are commonly used for model-checking purposes, because properties of specific such machines can be proved merely by trying all possibilities. Most papers about this assume that a finite-state machine is present as input. However, some work at a higher level of abstraction. In [32], Manna and Wolper consider the use of temporal logics (which have a long history in philosophy) as a model of the synchronization behaviour of a program, and show that in a specific temporal logic, satisfiability is decidable, and leads to finite models, which can then have implementations extracted from them. This last step is the unusual one; the finite model property itself was already familiar by 1981 (as Urquhart explains in [48]), but proofs of the property for various well-known logics (e.g. by Lafont in [30] for affine linear logic) typically see the property as a method of proving decidability. It may be possible to use this sort of approach to extract implementations from such languages, but this is a different approach than the one seen in this thesis (in which the languages are designed, or at least repurposed, to reflect particular implementation methods). Additionally, the source languages for which this sort of thing has been studied tend to be quite restrictive for producing programs in.

An alternative approach is to consider existing techniques for compiling programs down to finite-state representations; this is particularly studied in the field of hardware synthesis (the automatic generation of hardware from software). The most directly relevant series of papers to the direction taken in this thesis are the Geometry of Synthesis series by Ghica (and later also myself) from [15] onwards; many results from these are discussed later in the thesis.

The other commonly seen approach is to generate the low-level output from low-level input programs. In particular, the programming language C is a common choice for implementation in hardware. C can trivially be made to be finite-state by banning recursion (including indirectly), memory allocation, and higher-order functions; these restrictions appear in practice in some guidelines for safety-critical code (such as in Holzmann's report [25] on one set of rules used to develop software for NASA), and although these programs are designed to run in software, a static bound on the maximum amount of memory that can be used is important. Restrictions of C intended for compilation into hardware often have even stronger restrictions; although information on the exact implementations is often a commercial secret, there are frequent failures of abstraction, such as the timing restrictions in Handel-C that require all loops to contain at least one time-consuming command, as Celoxica explain in [6]. Probably the most common restriction is to have no support for functions other than inlining, as is the case in systems such as ROCCC (which Buyukkurt et al analyze in [5], discovering that the abstractions are inefficient enough that small amounts of loop unrolling can actually reduce the size of the output).

These languages fit in with the main theme of the thesis, in that the input languages are designed to reflect the limitations of the implementation techniques. However, the input languages in question tend to omit support for constructs like higher-order functions, and thus are much less powerful than the languages which are typically studied mathematically; this means that the main interest in them is in the implementation techniques, rather than the input languages. As such, looking at them in detail would be a distraction from the main goal of this thesis.

Part II

BACKGROUND MATERIAL

Chapter 2

MATHEMATICAL BACKGROUND

This thesis mentions several mathematical constructs that may not be familiar to everyone. This chapter briefly explains the key definitions, so that unfamiliar readers will be aware of the concepts in question before they are used in the main body of the text. The concepts here are standard, and readers familiar with them can safely skip this chapter.

2.1 Multisets

Sets and sequences are both familiar objects to mathematicians. It is possible to view a sequence as the mathematical object produced upon starting with a set, and then closing it under the most general possible associative binary operation that has a zero (in this thesis, this sequence concatenation operation is called $::$, and the zero, or empty sequence, is called ε); for example, a sequences of integers could be, say, $1::2::1$ (and because $::$ is associative by definition, $(1::2)::1$ and $1::(2::1)$ are equivalent). Likewise, the set of subsets of a set is produced via closing that set under the most general possible binary operation $\cup$ that is associative, commutative, has a zero $\emptyset$, and which discards duplicates (i.e. $x \cup x = x$). Because of the simple way in which sequences and sets can be generated from a binary operation, they frequently occur in syntax and semantics.

Multisets are a little less well-known as a mathematical object, and thus it seems prudent to introduce them before using them in this thesis; we will use them in Section 7.2 in order to represent a collection of time intervals. One way to look at a multiset (whose elements are elements of a given set) is as the mathematical object produced upon closing that set under the most general possible associative commutative binary operation which has a zero; in this thesis, the zero is called $[]$, and the operation $\uplus$. Multisets are somewhere between sets and sequences; multisets are like sets that care about how many copies they have of each element, or like sequences that do not care about which order they are in. The standard notation for multisets is similar to that for sets, but using square instead of curly brackets; for example, $[1,3,3,4]$ is a multiset which contains 1 and 4 once, and 3 twice (and thus is equal to $[3,1,4,3]$, but not to $[1,3,4]$; contrast this with sequences, where $3::1::4::3 \neq 1::3::3::4$, and sets, where $\{1,3,3,4\} = \{1,3,4\}$).

We give the following formal definition of multisets, in terms of functions:

Definition 2.1.1. A *multiset M* whose elements are taken from a set S is a function from S to $\mathbb{N}$ (where for each $s \in S$, s is said to be *contained $M(s)$ times* in the multiset). The *empty multiset* $[]$ is the function $\forall s.M(s) = 0$. We define the following operations on multisets: $(M \cap N)(s) \triangleq \min(M(s), N(s))$ (intersection); $(M \cup N)(s) \triangleq \max(M(s), N(s))$ (union); and $(M \uplus N)(s) \triangleq M(s) + N(s)$ (disjoint union), together with membership ($s \in M$ iff $M(s) > 0$).

When multisets are used in this thesis, it is typically to track some sort of object that can be used multiple times, perhaps with different properties each time: this is often convenient to track as a single object, and a multiset of properties. (Multisets are typically much simpler than sets for this purpose due to the fact that they do not disregard duplicates.)

2.2 Semirings and semigroups

Another mathematical object that frequently comes up in this thesis (for example, as the contraction bounds used in SCC) is the set of nonnegative integers, $\mathbb{N}$. There are two particularly important operations on integers: addition $+$ (which has an identity 0), and multiplication $\times$ (which has an identity 1). Sometimes, as with SCC, there is a good reason to use the integers specifically. On other occasions, however, all we actually care about is that addition and multiplication work the same way as with the integers.

Definition 2.2.1. A *semigroup* is a set equipped with a binary operation (which can have various names, typically $\odot$, $\oplus$, or $\otimes$) which is associative (i.e. $j \otimes (k \otimes l) = (j \otimes k) \otimes l$), and which has an identity (typically called $\mathbf{e}$, $\mathbf{0}$, or $\mathbf{1}$). This is both a left and right identity: $\mathbf{e} \otimes j = j = j \otimes \mathbf{e}$.

The nonnegative integers can be seen to be a semigroup in two different ways (via addition, and via multiplication), and have various other properties that relate these two semigroup operations. This pattern is one that is generally useful:

Definition 2.2.2. A *semiring* is a set which is a semigroup with operation $\oplus$ and identity $\mathbf{1}$, and also a semigroup with operation $\odot$ and identity $\mathbf{0}$. These are related by three further restrictions: the *distributivity* restrictions $j \odot (k \oplus l) = (j \odot k) \oplus (j \odot l)$ and $(j \oplus k) \odot l = (j \odot l) \oplus (k \odot l)$, and the rule that $j \odot \mathbf{0} = \mathbf{0} = \mathbf{0} \odot j$. Additionally, $\oplus$ must be commutative (i.e. $j \oplus k = k \oplus j$). (There is no restriction that $\odot$ must be commutative, and it frequently fails to be in practice.)

In this thesis, semirings normally become relevant due to the Application and Contraction rules found in many type systems. An Application rule often requires a multiplication-like operation, because if a function uses its argument multiple times, then that argument uses an inner argument multiple times, the number of uses need to be multiplied together. Contraction likewise needs an addition-like operation: if each of two variables can be used multiple times, then in order to merge them into a single combined variable, we need to add the number of uses. Therefore, it is frequently important for the mathematical objects we look at to have a meaningful addition equivalent $\oplus$ and multiplication equivalent $\odot$.

2.3 Categories

Category theory is a framework that describes results in many different fields of mathematics. Perhaps unsurprisingly, it can thus also be used to describe many results of type theory. For the benefit of unfamiliar readers, and to fix the notation, the basic definitions are listed here.

The basic definition of category theory is that of a *category*; this consists of *objects* and *morphisms*. Each morphism has two endpoints, going “from” one object “to” another object;

this is written in this thesis as "$f : A \longrightarrow B$" to mean "f is a morphism from the object A to the object B". There are only two further requirements on a category:

Definition 2.3.1. A collection of objects and morphisms form a category if: for any two morphisms $f : A \longrightarrow B$ and $g : B \longrightarrow C$, there is a morphism $f; g : A \longrightarrow C$ ($f; g$ is called the *composition* of f and g), with ; associative (i.e. $(f; g); h = f; (g; h)$); and for each object A, there is an *identity morphism* $\mathbf{id}_A : A \longrightarrow A$ such that given $f : A \longrightarrow B$, $\mathbf{id}_A; f = f = f; \mathbf{id}_B$.

Categories typically also have additional structure:

Definition 2.3.2. A *functor* is a function from the morphisms and objects of one category to the morphisms and objects of another that preserves object/morphism status, morphism endpoints, identities and composition; that is, for a functor $F : \mathscr{C} \to \mathscr{D}$, we have $F(f) : F(A) \longrightarrow F(B)$ whenever $f : A \longrightarrow B$, $F(\mathbf{id}_A) = \mathbf{id}_{F(A)}$, and $F(f; g) = F(f); F(g)$.

Notationally, this thesis uses a shorter arrow between the domain and codomain of a functor than it does between the endpoints of a morphism. Sometimes we will want to write down a functor directly as a pair of functions; an example of the notation, which simply shows the effect on objects and on morphisms, is $(A \mapsto A, f \mapsto f)$ for the identity functor.

One important categorical concept remains:

Definition 2.3.3. A *natural transformation* is a function α from objects of one category $\mathscr{C}$ to morphisms of a category $\mathscr{D}$ that "transforms between" two functors, $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{C} \to \mathscr{D}$, in the sense that given $f : A \longrightarrow B$, we have $F(f); \alpha(B) = \alpha(A); G(f)$ (which obviously requires $\alpha(A) : F(A) \longrightarrow G(A)$).

A natural transformation is most commonly thought of just as a family of morphisms which obeys certain properties; thus, the usual notation is α_A rather than $\alpha(A)$. (That is, $\mathscr{D}$ is the category we are really "interested in", and $\mathscr{C}$ exists to formalize the subscripts of our transformation α.)

The terminology "isomorphism" is also sometimes seen; an isomorphism is a morphism f that has an inverse f^{-1} such that $f; f^{-1}$ and $f^{-1}; f$ are both identities. A natural isomorphism is a natural transformation for which all the morphisms α_A are isomorphisms. We write $\alpha : F \rightarrow G$ as notation for a natural transformation, or $\alpha : F \cong G$ if it also happens to be a natural isomorphism.

Sometimes we want to be able to give a family of morphisms multiple subscripts, or have "variance" issues in a functor that reverse the direction of morphisms. This sort of problem is normally handled via a product and/or opposite category for the domain of the relevant functors:

Definition 2.3.4. The *product category* $\mathscr{C} \times \mathscr{D}$ of two categories $\mathscr{C}$, $\mathscr{D}$ has objects A, B consisting of an object A of $\mathscr{C}$ paired with an object B of $\mathscr{D}$; and morphisms $f, g : A, B \longrightarrow C, D$ consisting of a morphism $f : A \longrightarrow C$ of $\mathscr{C}$ and a morphism $g : B \longrightarrow D$ of $\mathscr{D}$. $\mathbf{id}_{A,B}$ is defined as $\mathbf{id}_A, \mathbf{id}_B$; and $f, g; f', g'$ is defined as $(f; f'), (g; g')$.

Definition 2.3.5. The *opposite category* $\mathscr{C}^{op}$ of a category $\mathscr{C}$ is $\mathscr{C}$ with the endpoints of each morphism swapped; $\mathbf{id}_A$ is the same in $\mathscr{C}$ and $\mathscr{C}^{op}$ (because its endpoints are the same), and $f; g$ in $\mathscr{C}^{op}$ is $g; f$ in $\mathscr{C}$.

The main use of category theory in this thesis is as a method of formalizing a denotational semantics; such a semantics normally forms a category where the denotations of types are objects, and the denotation of terms are morphisms between those objects. (This can be seen as comparable to the frequently-used category **Set** whose objects are sets, and whose morphisms are functions between those sets; however, a denotational semantics is typically more complex than this, mostly due to the existence of open terms.)

There is also an alternative commonly used notation for categorical statements, which consists of directed graphs where the vertices are labeled with objects, the edges with morphisms, and for which the morphism composition along any two paths between the same two vertices is equal. Some categorical statements in this thesis are presented using diagrams in addition to writing down the formulas, for the benefit of people who are familiar with this notation. However, no result is presented solely in diagram form, and thus there is no need to know this notation to read the thesis.

Chapter 3

TYPE SYSTEMS

In order to study programming languages mathematically, the first step is to be able to define a programming language mathematically. The description of a programming language comes in multiple components. Perhaps the most obvious is the language's syntax, defining which programs appear "well-formed". In fact, we work at a level slightly more general than a program; we normally work at the level of *terms*, which could represent a program as a whole, but also an expression or subexpression, statement, subroutine, function, or similar concept. Terms are divided into *closed terms*, which can be evaluated with no extra context (although closed terms that represent functions will require arguments before they can be evaluated), and *open terms*, which contain *free variables* which are merely placeholders until more context is provided. For example, $x+2$ is an open term in many programming languages, with x its only free variable, and the meaning of that expression is going to depend on what it is that x means in context. The large advantage of working with terms in general rather than with programs specifically is that it becomes possible to prove results about (or run algorithms on) the subterms of an term, and then inductively generalize that to the term as a whole, meaning that results about an entire program can be built up recursively as the sum of their parts.

In general, the details of the syntax of a language are mostly unimportant in terms of producing results about the language; for example, it doesn't matter whether a pair of two free variables is written in a form designed for input to a computer such as "`(x, y)`", or in a more mathematical notation like "$\langle x,y \rangle$". Likewise, in a practical programming language like Algol 60, a function definition could look something like "`integer procedure add2(x); integer x; begin add2 := x+2 end add2;`". The mathematical notation for this function could be much shorter, e.g. "$\mathit{add2} \triangleq \lambda x.\mathsf{op}_{+}(\langle x,2 \rangle)$". The difference is unimportant, however, because it is trivial to mechanically translate from one syntax to another. What is important, however, is that we fix a syntax so that it is possible to talk about it; in this thesis, we use the typical mathematical syntax for terms both because it is more concise and because it is easier to read. (For the benefit of readers unfamiliar with the syntax, it will be explained when first introduced.)

However, the syntax of a language is not nearly enough to define the language itself; a program can be nonsensical in ways other than failing to match the language's syntax. A term like "$\langle 1,2 \rangle \div 3$" makes no sense in many languages, because division of a pair of an integers, and a single integer, is not an operation that is compatible with the typical definition of division. Most practical language implementations have multiple components; first a syntax checker, which will typically have no problem parsing the term (so the implementation will understand that that program contains a division, and what is being divided by what), and secondly a type checker, which will complain that division cannot handle those operands. Mathematically, this term is rejected in a similar way; we define a *type system*, a set of rules for determining whether the term is correctly typed or not (and if it is, what types it can have). This chapter is focused on explaining the workings of type systems in detail, first in general (using simple examples), and then for some specific pre-existing type systems that are relevant to the thesis. (More type systems, created as part of the research into this thesis, will be presented later on.)

In addition to a language's type system, it also has a *semantics* (a set of rules for determining what it is that a program means); and practical languages typically also have one or more *implementations*, which make it possible to execute a program in practice. We will consider these in the next chapter.

3.1 Type judgements and derivations

As with so many other mathematical systems, the usual form for a type system consists of a set of axioms and a set of inference rules; a *derivation* starts with axioms, then applies rules to them in order to derive more complex terms. Although we are trying to determine what terms are legal, it is more useful for type systems to operate on a slightly different level; instead of axioms being terms, an axiom is a *type judgement* (or family of judgements) that is considered to be unconditionally derivable by the type system, and an inference rule says that a judgement is derivable if specific other judgements are derivable (and because an inference rule can require multiple judgements to be derivable, a derivation takes the form of a tree). A judgement is a statement that states specific circumstances under which a specific term has a specific type. Most type systems use judgements with the basic structure $\Gamma \vdash M : \theta$ (pronounced "M has type θ in a context where Γ"), and which consist of three parts: M is the term being judged; θ is the type that M is being judged to have; and Γ represents information about a context that allows that term to have that type (and which is normally expressed via giving restrictions on free variables). This gives an immediate way to define the difference between a closed and open term; an open judgement is one where Γ is nonempty, whereas a closed judgement has an empty Γ (and thus does not require any free variables to exist), and a closed term is a term for which a closed judgement can be constructed. (And in general, a term of the language as a whole is a term for which some judgement exists that is derivable within the type system, i.e. some derivation exists with that judgement at its root.)

As one of the simplest possible examples, consider a language which only has one type: the function (which takes a single function as an argument, and returns a function as its return value, because there are no other types for it to take or return). (We do not present proofs for statements made about the examples in this section, because they are used only as examples and none of the statements are used elsewhere in the thesis.) The syntax of this language (*untyped lambda calculus*), originally introduced by Church in [7, page 346] (with different notation), is shown in Figure 3.1.1: each term of the language is either a function parameter, a term given another term as an argument (what the notation MN means; note that parentheses need to be used to disambiguate terms like $M_1M_2M_3$ as either $M_1(M_2M_3)$ or $(M_1M_2)M_3$), or a function that takes one argument ($\lambda x.M$ means "a function that takes an argument x, and returns M", where M can contain references to the function parameter x). Sample terms of the language include the identity function, $\lambda x.x$, and the function that applies its argument to itself, $\lambda x.xx$. Note that

$$M ::= x \mid MN \mid \lambda x.M$$

Figure 3.1.1: Syntax of lambda calculus

$$\Gamma ::= \text{set of variable names}$$

$$\frac{}{\{x\} \vdash x : \mathsf{func}}\ \text{Identity}$$

$$\frac{\Gamma \vdash M : \mathsf{func}}{\Gamma \cup \{x\} \vdash M : \mathsf{func}}\ \text{Weakening}$$

$$\frac{\Gamma \cup \{x\} \vdash M : \mathsf{func} \qquad x \notin \Gamma}{\Gamma \vdash \lambda x.M : \mathsf{func}}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M : \mathsf{func} \qquad \Gamma \vdash N : \mathsf{func}}{\Gamma \vdash MN : \mathsf{func}}\ \text{Application}$$

Figure 3.1.2: Untyped lambda calculus (with contracting applications)

in our notation, M is used to stand for any term, and x for any variable name; so the identity function could just as easily have been written as $\lambda y.y$ (which is equivalent).

Writing a type system for this language may seem pointless; with only one type, all well-formed terms turn out to be derivable (even apparent nonsense like $\lambda x.y$, which is an *open* term that is perfectly meaningful if given some definition for y). However, when actually dealing with the language, it is easier to prove results using a more complex type system, so that it is possible to use induction on the derivation of the judgement of a term ("structural induction"); and even ignoring that, having a way to distinguish closed terms from open terms is useful.

As an example of what a typical type system might look like, Figure 3.1.2 shows one possible type system for untyped lambda calculus. Even though a much simpler type system (such as one which just had all the terms that matched the syntax as axioms) would be equivalent, this set of inference rules is a lot more useful for proving results in practice; suitably modified versions of many of the rules will turn up in other type systems later. The notation used for axioms and inference rules is the usual one; a judgement is written below the line, which is correctly typed if all the judgements above the line are correctly typed (and any side-conditions

$$\Gamma ::= \text{sequence of variable names}$$
$$x\#\Gamma \triangleq x \notin \Gamma$$

$$\frac{\nexists x. x \in \Gamma \wedge x \in \Delta}{\Gamma\#\Delta}$$

$$\frac{}{x \vdash x : \mathsf{func}}\ \text{Identity}$$

$$\frac{\Gamma \vdash M : \mathsf{func} \qquad x\#\Gamma}{\Gamma :: x \vdash M : \mathsf{func}}\ \text{Weakening}$$

$$\frac{\Gamma :: x :: y :: \Delta \vdash M : \mathsf{func}}{\Gamma :: y :: x :: \Delta \vdash M : \mathsf{func}}\ \text{Exchange}$$

$$\frac{\Gamma :: x :: \Delta \vdash M : \mathsf{func}}{\Gamma :: \Delta \vdash \lambda x.M : \mathsf{func}}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M : \mathsf{func} \qquad \Delta \vdash N : \mathsf{func} \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash MN : \mathsf{func}}\ \text{Application}$$

$$\frac{\Gamma :: x :: y \vdash M : \mathsf{func}}{\Gamma :: x \vdash M[x/y] : \mathsf{func}}\ \text{Contraction}$$

Figure 3.1.3: Untyped lambda calculus (with explicit contractions and ordered Γ)

written above the line hold); for an axiom, there are no judgements above the line; the notation for derivations is formed simply by stacking the *productions* (i.e. uses of an axiom or inference rule) on top of each other, with each rule's conditions being produced by previous rules. The type system allows useful statements to be proved about the language, such as "a term is closed if each of its variables x only appears inside a subterm starting with λx".

There is typically a lot of flexibility in writing a set of derivation rules for a type system. For example, Figure 3.1.3 shows a set of rules that are equivalent to the previous set, in that they describe the same terms, with the same types, and the same opinions on whether they are open or closed. However, it works somewhat differently; Figure 3.1.4 gives derivations for the same example term in both type systems. One major difference is that instead of free variable sets, it uses sequences instead, with a rule to allow them to be arbitrarily reordered (so that they act like sets). This might seem like overcomplicating things; however, it is nonetheless a step that is frequently taken when creating type systems, as it is one of the simplest ways to make the actual variable names irrelevant when proving results about the type system (instead

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{y\} \vdash y : \mathsf{func}}}{\{x,y\} \vdash y : \mathsf{func}} \quad \dfrac{\dfrac{}{\{x\} \vdash x : \mathsf{func}}}{\{x,y\} \vdash x : \mathsf{func}}}{\{x,y\} \vdash yx : \mathsf{func}} \quad \dfrac{\dfrac{}{\{y\} \vdash y : \mathsf{func}}}{\{x,y\} \vdash y : \mathsf{func}}}{\{x,y\} \vdash (yx)y : \mathsf{func}}}{\{x\} \vdash \lambda y.((yx)y) : \mathsf{func}}}{\emptyset \vdash \lambda x.\lambda y.((yx)y) : \mathsf{func}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y \vdash y : \mathsf{func}} \quad \dfrac{}{x \vdash x : \mathsf{func}}}{y :: x \vdash yx : \mathsf{func}}}{x :: y \vdash yx : \mathsf{func}} \quad \dfrac{}{z \vdash z : \mathsf{func}}}{x :: y :: z \vdash (yx)z : \mathsf{func}}}{x :: y \vdash (yx)y : \mathsf{func}}}{x \vdash \lambda y.((yx)y) : \mathsf{func}}}{\varepsilon \vdash \lambda x.\lambda y.((yx)y) : \mathsf{func}}$$

Figure 3.1.4: Example derivations in untyped lambda calculus

of using the names of the variables, you focus instead on their positions within the context). Another major difference is in the way multiple copies of the same variable are handled. With the first set of rules, an expression $\lambda x.M$ is handled by ensuring x is contained in the context of every subterm of M (because it could meaningfully be referenced from that location), meaning that Weakening has to be used in every subterm of M that *doesn't* mention x. With the second set of rules, the free variables are divided between sub-expressions in an application MN; to use the same variable in both M and N, it must first be given a temporary name in one of the expressions, and then the temporary name (z in the example) can be replaced to match the name in the other expression (y in the example), an operation known as *contraction* (and which will be the main focus of this thesis). ($M[x/y]$ means "M with all occurrences of y replaced by x".) Explicit contraction means that copying values is explicit, whereas implicit contraction means that copying values is implicit, and discarding the copies is explicit; thus, implicit contraction is simpler in cases where copying is "free", and explicit contraction when copying is something that needs to be reasoned about. This means that implicit contraction is more commonly used when discussing programming in general, but explicit contraction is more appropriate for the purposes of this thesis.

We also note some extra notation that will frequently be used throughout this thesis; often, side conditions in a type system will be quite complex, and hard to express within the individual rules, so they are moved to their own separate definitions. For example, Figure 3.1.3 uses a subsidiary definition # in order to express the notion of “freshness” (basically, avoiding clashes between variable names); these definitions can be placed on a single line using the defined-as notation “$\triangleq$” (e.g. the definition of $x\#\Gamma$), or as their own mini-type-systems (e.g. the definition of $\Gamma\#\Delta$). The freshness definition “#” will come up frequently in the type systems that follow, in various forms; many authors leave it implicit, but writing it explicitly is usually clearer and often also shorter (because explicit freshness restrictions avoid the need to explain what the implicit freshness rules are). One other frequently seen subsidiary definition is $\leq$, which is used for type systems where some sort of order exists on the types; untyped lambda calculus has only one type, so ordering its type is not particularly useful, but for some other type systems, an order on the types will be necessary when defining rules.

It is also possible to have type systems with the same syntax, but which produce noticeably different languages. For example, the type func is infinite due to being effectively equivalent to $\mathsf{func} \rightarrow \mathsf{func}$, and as such it might not exist in a practical programming language we are trying to model. We could instead disallow such types, and have only non-infinite types, where terms can have *base types* (that are not parameterized by other types, e.g. integers) or function types with specific arguments and return values (which thus disallows a function from being the same type as its argument or its return value). As is common in the field, we again restrict functions to have only one argument; to write a function with multiple arguments, it is simplest to accept a tuple as the argument, and even in type systems like this one without tuples or products, single-argument-single-return turns out to be enough due to the existence of functions that return other functions, as will be seen later. The resulting language is *typed lambda calculus*, which was also introduced by Church,[8, page 57] and a type system for it is shown in Figure 3.1.5. (Church’s original formulation was a little more complex, because it contained constants, which

$$\theta ::= \mathsf{exp} \mid \theta \to \theta$$
$$\Gamma ::= \text{sequence of } x\!:\!\theta$$
$$x\#\Gamma \triangleq \nexists\theta.(x\!:\!\theta) \in \Gamma$$

$$\frac{}{x\!:\!\theta \vdash x\!:\!\theta}\ \text{Identity}$$

$$\frac{\Gamma \vdash M\!:\!\theta \qquad x\#\Gamma}{\Gamma :: x\!:\!\theta' \vdash M\!:\!\theta}\ \text{Weakening}$$

$$\frac{\Gamma :: x : \theta' :: y : \theta'' :: \Delta \vdash M\!:\!\theta}{\Gamma :: y\!:\!\theta'' :: x\!:\!\theta' :: \Delta \vdash M\!:\!\theta}\ \text{Exchange}$$

$$\frac{\Gamma :: x\!:\!\theta' :: \Delta \vdash M\!:\!\theta}{\Gamma :: \Delta \vdash \lambda x.M\!:\!(\theta' \to \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M\!:\!(\theta' \to \theta) \qquad \Gamma \vdash N\!:\!\theta'}{\Gamma \vdash MN\!:\!\theta}\ \text{Application}$$

Figure 3.1.5: Typed lambda calculus

are discussed in Section 3.2; what is shown here is the most basic possible version.)

Although it has the same syntax, typed lambda calculus acts (with typical semantics) very differently from untyped lambda calculus. For example, there is no way to write an infinite loop, or recursion, in typed lambda calculus (at least as defined above). In untyped lambda calculus, simple infinite loops are easy to construct (such as $(\lambda x.xx)(\lambda y.yy)$) and recursive programs more generally can be constructed too, albeit with more difficulty, and with the details often depending on the details of the semantics.

3.2 Type system features

So far, we have looked at very simple type systems. In order to represent more complex languages, there are various other rules that are frequently added to the type system. One common feature of type systems is *constants*, a fixed set of closed terms that have their own syntax and act like axioms; these date back at least to the introduction of typed lambda calculus. One common use of constants is to represent the various values of the base types; as an example, if a type system has boolean as a base type, it typically has true : boolean and false : boolean as constants. Another use is to represent operators that exist in the language, which correspond to higher-order constants such as $\mathsf{and}:\mathsf{boolean} \to (\mathsf{boolean} \to \mathsf{boolean})$. This latter type might look unfamiliar to someone unused to functional programming, but is reasonably idiomatic; if a function would take two arguments, it can instead be implemented as a function that takes one argument, and returns a function which takes the other argument and returns the result. For example, $\mathsf{and}(\mathsf{true})(\mathsf{true})$ would be equivalent to true, and $\mathsf{and}(\mathsf{false})(\mathsf{true})$ to false. One sensible implementation of and would be for $\mathsf{and}(\mathsf{true})$ to be implemented equivalently to $\lambda x.x$, and $\mathsf{and}(\mathsf{false})$ to $\lambda x.\mathsf{false}$. This technique of implementing functions with multiple arguments as functions that return functions is known as *currying*, and was introduced by Schönfinkel in [43]. The Constant production is shown in Figure 3.2.1.

Some languages, however, have direct support for "tuples", the types of structures that contain a fixed number other values; for example, "one int and two floats" would be a tuple type. (Types such as the `struct` from C can be seen as a more user-friendly syntax for tuples.) In fact, because tuples can be nested, there is no real mathematical need for a language to support tuples with more than two elements. Tuples with one element are useless (being equivalent to the element itself), and although a tuple with zero elements is (perhaps surprisingly) not useless, all such tuples are clearly equivalent, and thus the zero-element tuple can be added as its own type (often called unit) if required. Thus, it makes sense to focus just on two-element tuples.

It might at first seem as though there is only one way to add a two-element tuple to a language. However, it will frequently be seen that there are actually two methods of implementing tuples, which are typically different in the languages that this thesis studies. In order to ensure that these are not confused, two different sets of notation will be used for the two sorts of tuple.

One commonly seen tuple implementation is the *product*. In this thesis, a product type is written as $\theta \times \theta'$; and the product constructor (i.e. a term that returns a product, specifying its left and right half directly) is written as $\langle M,N \rangle$. (Both these notations are standard.) If you have a product value, it is possible to (from that value) obtain either the left half of the product, or the right half of the product.

$$\Gamma \# \Delta \triangleq \nexists x.(\exists \theta.(x : \theta \in \Gamma)) \wedge (\exists \theta'.(x : \theta' \in \Delta))$$

$$\frac{M : \theta \text{ is a constant}}{\vdash M : \theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M : \theta \qquad \Gamma \vdash N : \theta'}{\Gamma \vdash \langle M, N \rangle : (\theta \times \theta')}\ \text{Product}$$

$$\frac{\Gamma \vdash M : \theta \qquad \Delta \vdash N : \theta' \qquad \Gamma \# \Delta}{\Gamma :: \Delta \vdash (M \otimes N) : (\theta \otimes \theta')}\ \text{Tensor}$$

$$\frac{\Gamma \vdash M : \theta \qquad \theta' \leq \theta}{\Gamma \vdash M : \theta'}\ \text{Subtyping}$$

Figure 3.2.1: Other rules commonly seen in type systems

Another commonly seen tuple implementation is the *tensor*. In this thesis, a tensor type is written as $\theta \otimes \theta'$; and the tensor constructor is written as $M \otimes N$. (Again, the use of $\otimes$ for tensors is standard.) If you have a tensor value, then you can obtain both the left half and right half of the tensor simultaneously, and operate on them at the same time.

We can see, then, that the difference between a product and a tensor is in what operations the underlying implementation supports. If you want to be able to access both halves of a product, you need to copy that product (because each copy of the product allows only one of its halves to be accessed); depending on the type system, this might be impossible (some type systems disallow copying entirely), or merely inefficient. Meanwhile, accessing both halves of a tensor is an operation that tensors specifically allow. On the other hand, there are typically fewer restrictions on constructing a product value than a tensor value. Figure 3.2.1 shows typical productions for creating products and for creating tensors; in many type systems, the $\Gamma \# \Delta$ restriction on creating a tensor is necessary (because otherwise, you could access the same free variable in parallel with itself by accessing both halves of the tensor at once, something that many type systems in this thesis are trying to disallow).

In order to provide access to the halves of a tuple, type systems normally provide constants. Products can be handled with a pair of constants $\pi_1 : (\theta \times \theta') \to \theta$ and $\pi_2 : (\theta \times \theta') \to \theta'$ that, when given a product as an argument, return the first or second half accordingly. These constants are called *projections*. Tensors are a bit more difficult; if you only provide projections, then you lose the benefits of having a tensor (because if the only way to unpack a tensor is to use projections, then you are restricted to the operations that you could perform on products). A good explanation of this phenomenon is given by O'Hearn et al in [35, page 7], where they discuss why they consider using projections to access the halves of a tensor in a preliminary version of their language was a mistake.

An example of a constant that can unpack a tensor while retaining its full power is $\mathsf{uncurry}: (\theta' \to (\theta'' \to \theta)) \to ((\theta' \otimes \theta'') \to \theta)$. Although it might look quite complex, it is conceptually simple, taking a curried function (i.e. one of type $(\theta' \to (\theta'' \to \theta))$) as its argument, and returning the equivalent tensor-based function (of type $((\theta' \otimes \theta'') \to \theta)$). Although the paper by O'Hearn et al is written in a form with a separate production for each language feature, rather than a single Constant production, their rule for unpacking a tensor is equivalent to $\mathsf{uncurry}$. It is possible to define the projections in terms of $\mathsf{uncurry}$, and thus although projections are meaningful on a tensor, a language that includes tensors rarely has a need to include projections explicitly, as they can be defined in terms of the rest of the language.

Because it is possible to define the projections on a tensor, a tensor can be seen as a more restrictive version of a product; because it supports all the same operations (with extra power, because it can also be uncurried), a tensor can thus be used in any context where a product is required. This is quite a common relationship in type systems in general. Several type systems thus have a *subtyping* relationship in which some types are considered to be special cases of others. These type systems have a Subtyping rule, shown in Figure 3.2.1, to allow the type of a term to be changed from a more specific type, to a more general type that contains it. (This relationship is typically written using $\leq$ notation; the definition of $\leq$ will depend on the type system.) Subtyping relationships are less common than constants, products, and tensors, but nonetheless come up often enough that they are worth mentioning here.

3.3 Type systems for programming

In this thesis, we are mostly focusing on what is necessary for a program (written in some programming language) to be implementable in the real world (either using a reprogrammable computer, or perhaps by generating a dedicated device for the purpose). Thus, we need to be able to represent a program mathematically. There are many type systems for programming languages in the literature; in this section, we look at some of the type systems that are most directly relevant to this thesis.

3.3.1 Idealized Algol

The Algol family of languages was very successful in the 1960s, and many of its features found its way into more modern languages that are still used today; for example, Algol popularized the "block structure" of compound statements and control flow statements that applied to them, which is very familiar to modern programmers from languages like Java and C. The most popular versions of Algol were Algol 60 (currently specified by its Modified Report, [2]), and later Algol 68 (currently specified by its Revised Report, [49]), which introduced many features to the language (such as heap allocation and parallel programming). Algol 60 and its variants are particularly suitable for this thesis for two reasons. One reason is that the most directly relevant existing results to this thesis are based on Algol 60 variants, and it makes more sense to be able to refer to existing results than start from scratch. The other is that Algol 60 uses an unusual "call-by-name" semantics of passing parameters to functions. This can be quite unfamiliar to programmers (and was actually abandoned in Algol 68 for this reason), but has several advantages for the purposes of this thesis. The main advantage is that it greatly reduces the number of special cases required in a type system; in the more common "call-by-value" semantics, a control structure like if needs to be a special case of its own, but in a call-by-name system, it can be implemented as a constant. It also produces a simpler semantics, in the sense that there is no need to worry about what a "function value" is; this expands the range of applications for the results in the thesis. Section 4.2 discusses this further.

$$\gamma ::= \mathsf{int} \mid \mathsf{real} \mid \mathsf{bool}$$
$$\theta ::= \gamma\,\mathsf{exp} \mid \gamma\,\mathsf{acc} \mid \gamma\gamma\,\mathsf{var} \mid \mathsf{com} \mid \mathsf{compl} \mid \mathsf{univ} \mid \theta \to \theta$$
$$\Gamma ::= \text{function from identifiers to } \theta$$

$$\frac{}{\mathsf{real} \le \mathsf{int}} \qquad \frac{}{\mathsf{univ} \le \theta} \qquad \frac{}{\mathsf{com} \le \mathsf{compl}} \qquad \frac{}{\gamma_1\gamma_2\,\mathsf{var} \le \gamma_1\,\mathsf{acc}} \qquad \frac{}{\gamma_1\gamma_2\,\mathsf{var} \le \gamma_2\,\mathsf{exp}}$$

$$\frac{\gamma \le \gamma'}{\gamma\,\mathsf{exp} \le \gamma'\,\mathsf{exp}} \qquad \frac{\gamma' \le \gamma}{\gamma\,\mathsf{acc} \le \gamma'\,\mathsf{acc}} \qquad \frac{\gamma_1' \le \gamma_1 \qquad \gamma_2 \le \gamma_2'}{\gamma_1\gamma_2\,\mathsf{var} \le \gamma_1'\gamma_2'\,\mathsf{var}}$$

$$\frac{M\!:\!\theta \text{ is a constant}}{\Gamma \vdash M\!:\!\theta}\,\text{Constant} \qquad \frac{\Gamma(x) = \theta}{\Gamma \vdash x\!:\!\theta}\,\text{Identity} \qquad \frac{\Gamma \vdash M\!:\!\theta \qquad \theta' \le \theta}{\Gamma \vdash M\!:\!\theta'}\,\text{Subtyping}$$

$$\frac{\{x \mapsto \theta', y \ne x \mapsto \Gamma(y)\} \vdash M\!:\!\theta}{\Gamma \vdash \lambda x.M\!:\!(\theta' \to \theta)}\,\text{Abstraction} \qquad \frac{\Gamma \vdash M\!:\!(\theta' \to \theta) \qquad \Gamma \vdash N\!:\!\theta'}{\Gamma \vdash MN\!:\!\theta}\,\text{Application}$$

Figure 3.3.1: Idealized Algol

One widely used mathematical formalization of Algol 60 is Reynolds's Idealized Algol, presented in [42]. The type system given in that paper, converted to a more modern notation, is given in Figure 3.3.1. Much of the complexity of the language comes from an interesting feature, abandoned in most of the other languages we look at: instead of having explicit deref or assign operators, it instead has a complex subtyping rule that allows variables to implicitly be used in the place of expressions, or of acceptors, and it allows a univ type (that can only be produced via the $\mathsf{case}_{0,\mathsf{univ}}$ constant that always errors out) that can be coerced into anything at all. An expression can produce a value; an acceptor can receive a value, and is typically produced via using subtyping on a variable. Idealized Algol also provides support for "goto" statements, via the use of a "completion" type compl that represents a jump to a label. The language's families of constants are those of Algol 60: constant integers (of type int exp), constant real numbers (of type real exp), true/false (of type bool exp), skip : com (which does nothing when executed), various arithmetic operators, $\mathsf{assign}_\gamma : \gamma\,\mathsf{acc} \to (\gamma\,\mathsf{exp} \to \mathsf{com})$ (which assigns an expression to an acceptor), $\mathsf{newvar}_{\gamma,\theta} : (\gamma\gamma\,\mathsf{var} \to \theta) \to \theta$ for $\theta \in \{\mathsf{com}, \mathsf{compl}\}$ (which binds a variable to a given scope), $\mathsf{seq}_\theta : \mathsf{com} \to (\theta \to \theta)$ for $\theta \in \{\mathsf{com}, \mathsf{compl}\}$ (which sequences commands), $\mathsf{while} : \mathsf{bool\,exp} \to \mathsf{com}$, $\mathsf{if}_\theta : \mathsf{bool\,exp} \to (\theta \to (\theta \to \theta))$, $\mathsf{case}_{j,\theta} : \mathsf{int\,exp} \to \overbrace{(\theta \to (\cdots \to (\theta \to}^{j\text{ times}}$

$\theta)\cdots)$ for $j \in \mathbb{N}$ (analogous to if but not limited to exactly 2 branches), $\mathsf{fix}_\theta : (\theta \to \theta) \to \theta$ (used to implement recursion), and $\mathsf{escape} : (\mathsf{compl} \to \mathsf{com}) \to \mathsf{com}$ (used to implement Algol's `go to` statement; $\mathsf{escape}(M)$ is equivalent to the Algol 60 code "`M(go to l1); l1:`", i.e. it declares a completion in much the same way that newvar declares a variable, and thus can be seen to be similar to the "call with current continuation" operation that exists in some other languages).

Idealized Algol captures Algol 60 quite well (there are some intentional deviations in order to correct infelicities in the definition of Algol 60), but there are some mathematical issues with this specific formalization. The formalization of contexts in terms of functions is quite hard to work with, and neither contraction nor weakening is explicit (the Weakening rule found in most type systems is implicit in the Identity rule of Idealized Algol). For this reason and others, Idealized Algol is mostly useful for historical context; in this thesis, we work with more modern type systems inspired by Idealized Algol, rather than working with Idealized Algol directly.

3.3.2 Idealized Concurrent Algol

Although Idealized Algol is a good starting point for this thesis, containing many features of interest to modern programmers (such as loops and variable declarations), it lacks one feature particularly relevant to this thesis. Entirely serial languages like Algol 60 or Idealized Algol only have one thread of execution, which allows for some huge simplifying assumptions: for example, execution is deterministic in the absence of constants added for the specific purpose of nondeterminism, and the only way a term can be executed multiple times simultaneously is via the use of terms of the shape $f(\ldots f(\ldots)\ldots)$. In practice, it is quite common to want to perform multiple calculations at the same time; and in this thesis, we aim for our results to be correct even in concurrent contexts.

Thus, it would seem useful simply to be able to add parallelism primitives to Idealized Algol. There are multiple possible ways to accomplish this, but the most relevant is the language introduced by Ghica and Murawski in [17] (which was not named in that paper, but which Ghica

$$\theta ::= \mathsf{com} \mid \mathsf{exp} \mid \mathsf{var} \mid \mathsf{sem} \mid \theta \to \theta \mid \theta \times \theta$$
$$\Gamma ::= \text{map from variables } x \text{ to types } \theta$$
$$x\#\Gamma \triangleq \Gamma \text{ does not map } x$$

$$\frac{}{x\!:\!\theta \vdash x\!:\!\theta}\ \text{Identity}$$

$$\frac{M : \theta \text{ is a constant}}{\vdash M\!:\!\theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M\!:\!\theta \qquad x\#\Gamma}{\Gamma, x\!:\!\theta' \vdash M\!:\!\theta}\ \text{Weakening}$$

$$\frac{\Gamma, x\!:\!\theta' \vdash M\!:\!\theta}{\Gamma \vdash \lambda(x\!:\!\theta').M\!:\!(\theta' \to \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M\!:\!(\theta' \to \theta) \qquad \Gamma \vdash N\!:\!\theta'}{\Gamma \vdash MN\!:\!\theta}\ \text{Application}$$

$$\frac{\Gamma \vdash M\!:\!\theta \qquad \Gamma \vdash N\!:\!\theta'}{\Gamma \vdash \langle M,N\rangle : (\theta \times \theta')}\ \text{Product}$$

Figure 3.3.2: Idealized Concurrent Algol with products

later called Idealized Concurrent Algol in [19], and which will often be referred to in this thesis via the abbreviated name ICA). Its type system (as opposed to the semantics) differs from an earlier attempt, Brookes's Parallel Algol ([3]), mostly in a different selection of constants (for example, ICA uses grab and release constants that atomically test-and-set single semaphores, whereas Parallel Algol uses a more complex await construct that can atomically test-and-set multiple Boolean variables at once); this thesis focuses on ICA because it is more similar to the languages in my existing results. ICA also simplifies Idealized Algol via removing extraneous features: there is no longer a distinction between integer/real/Boolean expressions; there are no goto statements; and variables are simpler, with explicit variable dereferencing, and with the observation that there is no reason to distinguish variables from acceptors.

The original paper on ICA does not implement any form of tuple. However, products can be added to the language straightforwardly, and future papers on the subject seem to assume that they exist, and thus in this thesis, "ICA" refers to a language that also contains products. Likewise, the original language had a separate production for each language feature, but it is more convenient to summarize most of them into a Constant rule. When written in this form, the ICA type system is shown in Figure 3.3.2.

The constants used are integers; $\mathsf{skip}:\mathsf{com}$ as in Idealized Algol; $\mathsf{op}_\bullet:\mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp})$ or $\mathsf{op}_\bullet:\mathsf{exp} \to \mathsf{exp}$, which represent arithmetic operations $\bullet$ (the exact type depends on $\bullet$, e.g. $\mathsf{op}_+:\mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp})$ but $\mathsf{op}_{\mathsf{succ}}:\mathsf{exp} \to \mathsf{exp}$; the list of operations is left unspecified, because the type system is parametric on it); $\mathsf{deref}:\mathsf{var} \to \mathsf{exp}$, variable dereferencing; $\mathsf{par}:\mathsf{com} \to (\mathsf{com} \to \mathsf{com})$ which runs two commands in parallel; $\mathsf{seq}_\theta:\mathsf{com} \to (\theta \to \theta)$ (for $\theta \in \{\mathsf{com}, \mathsf{exp}\}$), which runs two commands in series, or which runs a command before returning an expression; $\mathsf{assign}:\mathsf{var} \to (\mathsf{exp} \to \mathsf{com})$, which assigns a value to a variable; $\mathsf{newvar}_\theta:(\mathsf{var} \to \theta) \to \theta$ and $\mathsf{newsem}_\theta:(\mathsf{sem} \to \theta) \to \theta$ (for $\theta \in \{\mathsf{com}, \mathsf{exp}\}$), which create a variable or semaphore respectively in a given scope (a semaphore is basically a lock that can only be held by one grab operation at a time); $\mathsf{grab}:\mathsf{sem} \to \mathsf{com}$ and $\mathsf{release}:\mathsf{sem} \to \mathsf{com}$, synchronization primitives; $\mathsf{if}_\theta:\mathsf{exp} \to (\theta \to (\theta \to \theta))$ (for $\theta \in \{\mathsf{com}, \mathsf{exp}\}$), for conditionals; $\mathsf{fix}_\theta:(\theta \to \theta) \to \theta$, a recursion primitive; and the "bad constructors" $\mathsf{mkvar}:(\mathsf{exp} \to \mathsf{com}) \to (\mathsf{exp} \to \mathsf{var})$ and $\mathsf{mksem}:\mathsf{com} \to (\mathsf{com} \to \mathsf{sem})$, which allow the creation of variables and semaphores respectively with arbitrary reactions to assign/dereference or to grab/release, and which are sometimes needed when constructing terms that have a specific denotation (the original source for this was cited in [18] as being the full version of the paper by Abramsky and McCusker whose abstract is [1], but I have not been able to track down the full paper to verify this; a more detailed account of the need for mkvar is given by McCusker in [33, page 171]). Because we have added products to ICA, we also have the projections $\pi_{1,\theta,\theta'}:(\theta \times \theta') \to \theta$ and $\pi_{2,\theta,\theta'}:(\theta \times \theta') \to \theta'$, as usual.

In this thesis, we will typically be looking at restrictions of ICA in which there are restrictions on copying terms, which makes grab and $\mathsf{release}$ useless for synchronization because the semaphore that they synchronize on cannot be copied to multiple threads of execution, and thus we will normally consider the fragment of the language that does not contain semaphores. (A *fragment* of a type system is a type system that contains a subset of its rules, a subset of its constants, and which may place extra conditions on some of the rules. Any derivation legal in a type system fragment is clearly legal in the type system as a whole.) Likewise, we will typically ignore mkvar; partly this is because we are not trying to produce terms to match denotations (we go the other way, from terms to denotations), and partly this is because many of our type systems define var as a tuple of $\mathsf{exp} \to \mathsf{com}$ and exp rather than as a separate type in its own right, in which case mkvar can be trivially implemented using a tuple constructor.

ICA with products can be considered an "upper bound" on the type systems examined in this thesis; terms derivable in most of the type systems in this thesis are also derivable in ICA, and in general, the type systems aim to be ICA with added restrictions. One way to think about this thesis is that it aims to identify a range of useful finite-state subsets of ICA.

3.3.3 Basic Syntactic Control of Interference

This thesis mostly looks at the question of how to guarantee that a program is finite-state. The language Syntactic Control of Interference (or SCI) was originally created by Reynolds in [40] to guarantee a different property ("non-interference"): that no variable is written to while another part of the program is trying to access it. However, it turns out that non-interference and ensuring finite-state have something of a common theme, in that in both properties are trying to prevent simultaneous use of something (for non-interference, we want to prevent simultaneous use of a variable, and for a program to be finite-state, we want to prevent simultaneous use of a term, because each such use would need its own independent internal state). Thus, many of the techniques used in SCI happen to be relevant to this thesis as well. As a bonus, SCI and ICA are by the same author, meaning that the languages are similar and can easily be compared.

The full SCI language is quite complex, and contains functionality that is not relevant for this thesis. In [34, page 28], O'Hearn presented a cut-down version of SCI, called "Basic Syntactic Control of Interference" (often abbreviated to bSCI in this thesis), as an introduction to the SCI family of languages. In addition to making a good introduction, bSCI also turned out to be a useful language in its own right, and inspired various other relevant languages; this thesis discusses SCI-like languages in Chapter 8. Thus, bSCI is presented here both so that

$$\theta ::= \mathsf{exp} \mid \mathsf{var} \mid \mathsf{com} \mid \theta \multimap \theta \mid \theta \times \theta$$
$$\Gamma ::= \text{list of } x\colon\theta$$
$$\Gamma\#\Delta \triangleq \nexists x.(\exists \theta.x\colon\theta \in \Gamma) \wedge (\exists \theta'.x\colon\theta' \in \Delta)$$

$$\frac{}{x\colon\theta \vdash x\colon\theta}\ \text{Identity}$$

$$\frac{M : \theta \text{ is a constant}}{\vdash M\colon\theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M\colon\theta \qquad \Gamma \text{ is a permutation of } \Delta}{\Delta \vdash M\colon\theta}\ \text{Exchange}$$

$$\frac{\Gamma \vdash M\colon\theta \qquad x\colon\theta'\#\Gamma}{\Gamma :: x\colon\theta' \vdash M\colon\theta}\ \text{Weakening}$$

$$\frac{\Gamma :: x\colon\theta' \vdash M\colon\theta}{\Gamma \vdash \lambda(x\colon\theta').M\colon(\theta' \to \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M\colon\theta' \to \theta \qquad \Delta \vdash N\colon\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash MN\colon\theta}\ \text{Application}$$

$$\frac{\Gamma \vdash M\colon\theta \qquad \Gamma \vdash N\colon\theta'}{\Gamma \vdash \langle M,N\rangle\colon(\theta \times \theta')}\ \text{Product}$$

$$\frac{x\colon\theta \vdash M\colon\theta}{\mathsf{fix}\,\lambda x.M\colon\theta}\ \text{Recursion}$$

Figure 3.3.3: Basic Syntactic Control of Interference

the reader will understand the idea behind the SCI family of languages, and because it will be directly relevant later on.

Adjusting the notation from the original paper to the notation of this thesis (e.g. using var for the type of variables rather than cell like the original paper used), and combining as many language features as possible into a single Constant rule, produces the formalization of bSCI shown in Figure 3.3.3. The selection of constants is much the same as in ICA without semaphores (π_1, π_2, par, seq, integers, op, if, newvar, deref, and assign); in the original formalization of bSCI, deref (and nothing else) is implemented as subtyping, but this thesis implements it as a constant instead for clarity. However, there are three major differences. Application no longer allows implicit contraction. fix is no longer a constant, but instead has to be implemented as a type production (Recursion); the reasons for this are quite complex, and explained in Chapter 9. Finally, some types of constants are quite unusual: the largest change from ICA is that $\mathsf{par} : \mathsf{com} \to (\mathsf{com} \to \mathsf{com})$ (as in ICA) but $\mathsf{seq}_{\mathsf{com}} : (\mathsf{com} \times \mathsf{com}) \to \mathsf{com}$. (Most other constants are also written in curried form, like seq; par is an exception.)

The difference between par and seq reflects the main idea behind SCI, and that is reflected in bSCI: functions that potentially use their arguments in parallel are expressed in terms of currying, whereas if a function uses its arguments only sequentially, it can take a product as its argument instead. This means that there is a syntactic difference between concurrency and sharing, ensuring that they never occur at the same time. (Concurrency and sharing are not mutually exclusive in most languages; however, SCI aims to prevent interference via preventing sharing in concurrent contexts.) The choice is not arbitrary; nested application of a function (as in $f(f(x))$) can cause problems with sharing even in non-concurrent contexts (if f is stateful, then the "inner" copy of f could potentially overwrite state that the "outer" copy is using to decide what to do after it returns), and thus function application needs to disallow sharing (and thus the opposite choice, using currying for sharing and tensors for concurrency, would not work). This design feature is reflected in the Application rule, which requires a function and its argument to have disjoint free variables.

Many of my papers that predate this thesis ([19, 20, 22]) are based around hardware implementations of SCI-based languages (hardware is inherently finite-state, so the languages must be too). Thus, bSCI and friends seemed at first to answer the question posed by the thesis. As we will discover in Chapter 8, though, there are problems with this view. Nonetheless, bSCI and related languages are very important to this thesis, and understanding the idea behind them will make the thesis easier to follow.

Chapter 4

SEMANTICS AND IMPLEMENTATIONS

A type system expresses which terms are considered by a language to be meaningful. However, it does not explain what that meaning is. In order to get a complete mathematical description of a language, we thus need a description of what each term means, in addition to a description of which terms have meanings.

The normal mathematical way to describe the meaning of terms in a type system is by giving a *semantics* for that type system. There are two common forms of semantics: an *operational semantics* is an algorithm for evaluating the term (eventually producing some sort of mathematical object that can be thought of as describing the value of that term); and a *denotational semantics* is a function from terms to some set such that terms have the same behaviour if and only if they have the same "denotation" (i.e. image under that function).

There is another, less mathematical, way to describe the meanings of the term in a type system: you can *implement* that type system in some other language, or on a physical computer. An example would be a compiler, which translates one language to another; you can take a program in the original language and look at the compiler output in a different language. Implementations are not devoid of mathematical interest either; for instance, if two terms have identical denotations, then their implementations would also be expected to have identical denotations. However, their implementations would not necessarily be the same; for example, one might be more efficient than the other.

Semantics is not a major focus of this thesis, but it will discuss denotational semantics on occasion when proving results about the meaning of a term, and some idea of the semantics of a language is needed to guide the design of the language. This thesis also does not look at implementations in detail, but does briefly look at how a language feature might be implemented; most of the language features in the type systems discussed were inspired by features of implementations.

4.1 Operational and denotational semantics

Consider a typed lambda calculus term such as $(\lambda(x : \mathsf{int}).(\mathsf{op}_+ x)(x))(6)$. (We are selecting a simple term and language here in order to illustrate the concepts.) In order to work out the meaning of this term, we need definitions for what x, $\lambda x.M$ and MN mean. We also need a definition of the constant op_+.

x, $\lambda x.M$ and MN all have standard definitions. In a closed term, all variables x are *bound* (not free), so they get their meaning from a function application $(\lambda x.M)(N)$; this gives us a definition of all three lambda calculus constructs. Thus, the most obvious algorithm for evaluating such a term is to consistently replace all subterms of the form $(\lambda x.M)(N)$ with $M[N/x]$. (This particular rule for evaluating lambda calculus and related type systems is standard, and known as "β-reduction".) Each constant also needs its own rule; in this case, we can reasonably define $(\mathsf{op}_+ j)(k)$ as $j + k$ for all integer constants j and k. We can see what happens to our example term in Figure 4.1.1a; it eventually reduces to 12, as expected. (Note that we also need to consistently replace variable names in N when doing the substitution, to avoid clashes.) An operational semantics is typically formed of rules such as these.

One way to assign meanings to terms, then, is to apply the rules of an operational semantics to them until we end up with a term that does not reduce further. We can consider all terms that appear in the reduction to have the same meaning; thus, the term we eventually end up with (a *normalized* term) can be thought of as representing the term's behaviour in some way.

An alternative method is to associate a semantics with a judgement via structural induction: we can define a semantics for judgements produced by the Identity rule and each constant, then define the semantics for each other judgements via combining the semantics of the judgements

$$(\lambda x.M)(N) \longrightarrow M[N/x] \text{ with fresh variable names in substitutions of } N$$

$$\begin{aligned}
&(\lambda(x\!:\!\mathsf{int}).(\mathsf{op}_+ x)(x))(6)\\
&(\mathsf{op}_+ 6)(6)\\
&12
\end{aligned}$$

(a) Operational semantics (β-reduction)

$$\begin{aligned}
[\![\theta]\!] &: \mathbb{N}\\
[\![\mathsf{int}]\!] &\triangleq 0\\
[\![\mathsf{int} \to \theta]\!] &\triangleq [\![\theta]\!] + 1\\
[\![x_1 :: \ldots :: x_j]\!] &\triangleq j
\end{aligned}$$

$$\begin{aligned}
[\![\Gamma \vdash M\!:\!\theta]\!] &: \mathbb{Z}^{[\![\Gamma]\!]+[\![\theta]\!]} \to \mathbb{Z}\\
[\![Id_x : (x \vdash x\!:\!\mathsf{int})]\!] &\triangleq \langle j \rangle \mapsto j\\
[\![Abs_x(\nabla) : (\Gamma \vdash \lambda x.M\!:\!\mathsf{int} \to \theta)]\!] &\triangleq [\![\nabla : (\Gamma :: x \vdash M\!:\!\theta)]\!]\\
[\![App(\nabla,\nabla') : (\Gamma :: \Delta \vdash MN : \theta)]\!] &\triangleq \langle j_1, \ldots, j_{[\![\Gamma]\!]+[\![\Delta]\!]+[\![\theta]\!]} \rangle \mapsto\\
&\qquad [\![\nabla : (\Gamma \vdash M : \mathsf{int} \to \theta)]\!]\\
&\qquad\quad (\langle j_1, \ldots, j_{[\![\Gamma]\!]},\\
&\qquad\quad\ \ [\![\nabla' : (\Delta \vdash N\!:\!\mathsf{int})]\!](\langle j_{[\![\Gamma]\!]+1}, \ldots, j_{[\![\Gamma]\!]+[\![\Delta]\!]} \rangle),\\
&\qquad\quad\ \ j_{[\![\Gamma]\!]+[\![\Delta]\!]+1}, \ldots, j_{[\![\Gamma]\!]+[\![\Delta]\!]+[\![\theta]\!]} \rangle)\\
[\![Contr_{x,y}(\nabla) : (\Gamma :: x \vdash M[x/y] : \theta)]\!] &\triangleq \langle j_1, \ldots, j_{[\![\Gamma]\!]+[\![\theta]\!]+1} \rangle \mapsto\\
&\qquad [\![\nabla : (\Gamma :: x :: y \vdash M : \theta)]\!]\\
&\qquad\quad (\langle j_1, \ldots, j_{[\![\Gamma]\!]+1}, j_{[\![\Gamma]\!]+1}, \cdots j_{[\![\Gamma]\!]+[\![\theta]\!]+1} \rangle)\\
[\![Const_{\mathsf{op}_+} : (\vdash \mathsf{op}_+ : \mathsf{int} \to (\mathsf{int} \to \mathsf{int}))]\!] &\triangleq \langle j,k \rangle \mapsto j+k\\
[\![Const_j : (\vdash j\!:\!\mathsf{int})]\!] &\triangleq \langle\rangle \mapsto j
\end{aligned}$$

$$\begin{aligned}
[\![\vdash \mathsf{op}_+ : \mathsf{int} \to (\mathsf{int} \to \mathsf{int})]\!] &= \langle j,k \rangle \mapsto j+k\\
[\![x \vdash x\!:\!\mathsf{int}]\!] &= \langle l \rangle \mapsto l\\
[\![x \vdash \mathsf{op}_+ x\!:\!\mathsf{int} \to \mathsf{int}]\!] &= \langle l,k \rangle \mapsto l+k\\
[\![y \vdash y\!:\!\mathsf{int}]\!] &= \langle m \rangle \mapsto m\\
[\![x,y \vdash (\mathsf{op}_+ x)(y)\!:\!\mathsf{int}]\!] &= \langle l,m \rangle \mapsto l+m\\
[\![x \vdash (\mathsf{op}_+ x)(x)\!:\!\mathsf{int}]\!] &= \langle l \rangle \mapsto l+l\\
[\![\vdash \lambda x.(\mathsf{op}_+ x)(x)\!:\!\mathsf{int} \to \mathsf{int}]\!] &= \langle l \rangle \mapsto l+l\\
[\![\vdash 6\!:\!\mathsf{int}]\!] &= \langle\rangle \mapsto 6\\
[\![\vdash (\lambda x.(\mathsf{op}_+ x)(x))(6)\!:\!\mathsf{int}]\!] &= \langle\rangle \mapsto 12
\end{aligned}$$

(b) Denotational semantics

Figure 4.1.1: Semantics of $(\lambda(x\!:\!\mathsf{int}).(\mathsf{op}_+ x)(x))(6)$

it is derived from. Strictly speaking, we are assigning semantics to derivations ∇; however, it is usual to define a semantics such that all derivations of the same judgement have the same semantics. This is a denotational semantics; we use the notation $[\![\nabla]\!]$ (or often $[\![\nabla : (\Gamma \vdash M : \theta)]\!]$ for more clarity) to represent the denotation of a derivation ∇ that derives a judgement $\Gamma \vdash M : \theta$. Denotational semantics can often be quite complex; Figure 4.1.1b shows a denotational semantics for a fragment of typed lambda calculus (the fragment in which types are restricted to $\theta ::= \mathsf{int} \mid \mathsf{int} \to \theta$ and there is no Exchange rule). We have semantics for each type $[\![\theta]\!]$, in addition to each judgement. The semantics can be arbitrary mathematical objects so long as two terms have the same denotation if and only if they mean the same thing; in this case, types are denoted by nonnegative integers, and judgements by functions from tuples of integers to integers.

Both these semantics give suggestions for implementing this language. The operational semantics suggests that the language could be implemented by a string-rewriting machine, whereas the denotational semantics is quite reminiscent of some sorts of stack machine. In practice, an implementation of this language would be unlikely to use either approach directly; most likely, it would do some amount of rewriting to uncurry functions, and might perhaps use something like a stack machine from there.

4.2 Evaluation order

One important decision that is typically invisible in a type system itself, but can be very noticeable in a semantics, is to do with the order in which subterms are evaluated/reduced. In a language in which all programs terminate, all evaluation is deterministic, and there are no effects (neither side effects nor intended effects of constants), the evaluation order generally does not matter. However, many languages contain constructs such as assignment to variables. (There is an unfortunate terminology clash here. “Free variable” and “bound variable” refer to the identifiers used in a term M or context Γ, as does “variable” in most cases. “Assignable variable” can be used to refer to values of type var, that typically refer to modifiable memory locations, and are typically created by constants like newvar; these are just called "variables" in most programming languages, and mathematical discussions also sometimes use that terminology when the meaning is clear from context.) Some sort of evaluation order then needs to be fixed. Otherwise, an operational semantics may produce different results for the same term depending on how it is applied, and a denotational semantics will be impossible to construct.

There are numerous possible evaluation orders that could be specified. However, this section will only mention two of the most common. The best-known evaluation order is *call-by-value*, an evaluation order in which the argument to a function is fully evaluated before the application of that argument to that function is expanded. Thus, in a call-by-value semantics, a term such as $\mathsf{newvar}(\lambda x.((\lambda y.\mathsf{deref}(x))(\mathsf{assign}(x)(1))))$ will create a variable x, then evaluate $\mathsf{assign}(x)(1)$ (the argument to $\lambda y.\mathsf{deref}(x)$), then throw away the resulting value (y is unused by that function), and return the value currently in x. This will be 1, the value that was assigned by the argument to the function.

This points to a problem with call-by-value. In such languages, any side effects to the argument to a function will happen before the function is called, meaning that it is impossible to express constructs like an imperative if as a constant; it would have no way to avoid evaluating one of its arguments. while is an even more dramatic example; because in call-by-value its test expression is evaluated just once before the constant itself, each iteration of the loop is using the value that the test expression had at the start of the loop. Thus, either the test expression evaluated to false and the loop is skipped entirely, or it evaluated to true, forcing the loop to repeat forever (which is not particularly useful if you want the possibility of your program halting). Another illustration of the same basic problem is that all possible com-typed arguments to a function are equivalent from the point of view of that function (in fact, in such languages, com is commonly implemented as a zero-element tuple, because when receiving it as an argument it contains no information). The normal workarounds to this problem include adding special cases to constants to change their evaluation order, and giving terms dummy lambda arguments in order to delay their evaluation. Call-by-value languages are thus not considered further in this thesis, although it would not surprise me if many of the results were nonetheless applicable (perhaps with modifications) in a call-by-value context.

One of the main alternatives to call-by-value is known as *call-by-name*; this was most famously used by Algol 60, but is frequently seen in mathematical languages nowadays. The rule here is that a subterm cannot be expanded if it is, or is inside, the argument side of any application; evaluation proceeds by evaluating the function side of the application (if necessary), then using the argument as-is. Thus, arguments to functions are only expanded at the last possible moment, or not at all if they are unused. The largest advantage to this method is that there is no problem with using constants for if, while and friends. It can confuse programmers who are used to call-by-value, though; an example is $x : \mathsf{var} \vdash (\lambda y.\mathsf{seq}(\mathsf{assign}(x)(2))(y))(\mathsf{op}_{-}(\mathsf{deref}(x))(1))$ (which may be easier to read in a less mathematical syntax, such as that discussed in Chapter 10: `(\y.x:=2; y)(!x-1)`), which always evaluates to 1 regardless of the old value of x (because by the time the $(\mathsf{op}_{-}(\mathsf{deref}(x))(1))$ is evaluated, x already has the value 2, and so the $x-1$ subtraction never happens with the old value of x).

These calling conventions also tend to lead to rather different implementations. A call-by-value implementation typically has a concept of values for each type: integer values, function values, etc.. The value of an integer is normally trivial to implement. The value of a function can be much harder to implement, however (one common technique is the use of a *closure*, which contains a reference to code implementing the function, plus a copy of the values that any bound variables in that function were bound to). In a call-by-name implementation, functions instead take terms (or the compiled representation of terms) as their arguments; thus, there is no need for anything like a closure, and values exist only as they are being used, rather than being stored. This also points to the main disadvantage of call-by-name: in order for a value to be reused (rather than recalculated each time it is needed), it must be stored in an assignable variable. Still, the advantages are large enough that this thesis typically designs type systems to work with a call-by-name evaluation order.

Part III

FINITE-STATE TYPE SYSTEMS

Chapter 5

TYPE SYSTEMS WITHOUT CONTRACTION

At the heart of the problem of infinite-state programs are the concepts of sharing and copying. It is common in the semantics of a type system to be able to make arbitrarily (or even infinitely) many copies of something, perhaps in response to the control flow of the program, and run them in a nested or even parallel fashion. For example, in typical semantics for ICA, composing a function with its argument may require making multiple copies of the argument, and for functions like par, those arguments may run in parallel. Clearly, eliminating sharing and copying from a language altogether is a simple way to make it impossible to construct an infinite program: because no part of the original program can be used multiple times, the implementation must be finite because the original program is finitely long.

Idealized Concurrent Algol is a good representation of the sort of language we would like to generate practical implementations of: it is based on a language (Algol 60) that was widely used in practice (and thus likely to be practically useful), and it has powerful features like concurrency and higher-order functions. However, it allows unlimited sharing of free variables. In this chapter, we explore one of the simplest ways to ensure a language is finite-state; removing contraction from the type system altogether (to produce a language "Affine ICA"). Because the need to copy a function's argument is essentially caused by contraction of free variables, removing the feature altogether ensures that each term can only be evaluated once, ever. (We also remove recursion, another language feature that could lead to a term being evaluated more than once; some restricted versions of recursion will be reintroduced into some of the languages we consider in later chapters.) This leads to a type system that is not particularly practically useful: disallowing terms from being evaluated more than once means that even tail recursion is disallowed, and as such it is impossible to write any form of loop. It is still useful, however, to consider such a type system in detail: most of the techniques that apply to Affine ICA are still relevant even when looking at more complex languages. Such languages are interesting mostly in how they differ from Affine ICA, meaning that knowing about the base language will allow us to focus only on the differences, without having to also consider the parts that remain the same.

5.1 The Affine ICA type system

Figure 5.1.1 shows an ICA type system, this time written using the standard transformation that expresses a type system using an explicit Contraction rule. As usual, an ICA term is any M for which there is a Γ and θ such that $\Gamma \vdash M : \theta$ has a derivation. The difference from the usual statement of ICA is that the application rule and pair formation rule now have an additional restriction that prevents the free variables in the two free variable lists from sharing, and an extra contraction rule has been added that allows arbitrary identifiers to be contracted. There is a simple mechanical translation between a derivation in the usual statement of ICA, and in this explicit version: two identifiers with the same name (but in the free variable lists of different terms) correspond to two identifiers with different names, that are then contracted. An example of the correspondence is shown in Figure 5.1.2; in the first derivation, the same variable name x can be used because the two sides of an application use the same free variable list, and in the second derivation, two variable names x and y are used because they can be contracted into a single variable after the application is performed.

Because this correspondence between implicit and explicit contraction is necessary to understand most of this thesis, it's worth seeing what the proof of the correspondence typically looks like:

Theorem 5.1.1. *The set of terms that can be derived via the ICA type system in Figure 3.3.2 is*

$$\theta ::= \mathsf{com} \mid \mathsf{exp} \mid \mathsf{var} \mid \mathsf{sem} \mid \theta \to \theta \mid \theta \times \theta$$
$$\Gamma ::= \text{sequence of } x\!:\!\theta$$

$$\frac{x \neq y}{x\!:\!\theta \# y\!:\!\theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma :: \Gamma'\#\Delta} \qquad \frac{x : \theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{x\!:\!\theta \vdash x\!:\!\theta}\ \text{Identity}$$

$$\frac{M : \theta \text{ is a constant}}{\vdash M\!:\!\theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M\!:\!\theta \qquad x\#\Gamma}{\Gamma :: x\!:\!\theta' \vdash M\!:\!\theta}\ \text{Weakening}$$

$$\frac{\Gamma :: x\!:\!\theta' :: y\!:\!\theta'' :: \Delta \vdash M\!:\!\theta}{\Gamma :: y\!:\!\theta'' :: x\!:\!\theta' :: \Delta \vdash M\!:\!\theta}\ \text{Exchange}$$

$$\frac{\Gamma :: x\!:\!\theta' :: \Delta \vdash M\!:\!\theta}{\Gamma :: \Delta \vdash \lambda(x\!:\!\theta').M\!:\!(\theta' \to \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M\!:\!(\theta' \to \theta) \qquad \Delta \vdash N\!:\!\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash MN\!:\!\theta}\ \text{Application}$$

$$\frac{\Gamma \vdash M\!:\!\theta \qquad \Delta \vdash N\!:\!\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash \langle M,N\rangle : (\theta \times \theta')}\ \text{Product}$$

$$\frac{\Gamma :: x\!:\!\theta' :: y\!:\!\theta' \vdash M : \theta}{\Gamma :: x\!:\!\theta' \vdash M[x/y] : \theta}\ \text{Contraction}$$

Figure 5.1.1: Idealized Concurrent Algol, with explicit contractions

Without explicit contractions:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{seq} : (\mathsf{com} \to (\mathsf{com} \to \mathsf{com}))} \qquad \dfrac{}{x\!:\!\mathsf{com} \vdash x\!:\!\mathsf{com}}}{x\!:\!\mathsf{com} \vdash \mathsf{seq}(x) : (\mathsf{com} \to \mathsf{com})} \qquad \dfrac{}{x\!:\!\mathsf{com} \vdash x\!:\!\mathsf{com}}}{x\!:\!\mathsf{com} \vdash \mathsf{seq}(x)(x) : \mathsf{com}}}{\vdash \lambda(x\!:\!\mathsf{com}).\mathsf{seq}(x)(x) : \mathsf{com}}$$

With explicit contractions:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{seq} : (\mathsf{com} \to (\mathsf{com} \to \mathsf{com}))} \qquad \dfrac{}{x\!:\!\mathsf{com} \vdash x\!:\!\mathsf{com}}}{x\!:\!\mathsf{com} \vdash \mathsf{seq}(x) : (\mathsf{com} \to \mathsf{com})} \qquad \dfrac{}{y\!:\!\mathsf{com} \vdash y\!:\!\mathsf{com}}}{x\!:\!\mathsf{com} :: y\!:\!\mathsf{com} \vdash \mathsf{seq}(x)(y) : \mathsf{com}}}{x\!:\!\mathsf{com} \vdash \mathsf{seq}(x)(x) : \mathsf{com}}}{\vdash \lambda(x\!:\!\mathsf{com}).\mathsf{seq}(x)(x) : \mathsf{com}}$$

Figure 5.1.2: Example of ICA with explicit contractions

the same as the set of terms that can be derived via the ICA type system with explicit contractions in Figure 5.1.1.

Proof sketch. (Derivations in this proof sketch are given using a notation that treats production rules like functions.) Given an ICA derivation with implicit contractions, we can make a derivation with explicit contractions via the following replacements (where $x_1 \dots x_j$ are the free variables of the term derived by ∇_1, and $y_1 \dots y_j$ are fresh variables):

$$App(\nabla_1, \nabla_2) \mapsto Contr(x_1, y_1, Contr(\dots, Contr(x_j, y_j, App(\nabla_1, \nabla_2[y_k/x_k]))\dots))$$

$$Prod(\nabla_1, \nabla_2) \mapsto Contr(x_1, y_1, Contr(\dots, Contr(x_j, y_j, Prod(\nabla_1, \nabla_2[y_k/x_k]))\dots))$$

That is, each use of Application and Product is transformed via renaming all the variables on one side, and then contracting the old names with the new names. (Some uses of Exchange also need to be added for Contraction to apply, but it is clear that enough Exchanging can rearrange the sequences of free variables into any order.) Because all the implicit contraction rules other than Application and Product exist with explicit contractions too, this is the only change necessary.

To go the other way, the rules that need changing are Application, Product, and Contraction. First, any Contractions must be eliminated, via moving them up towards the leaves of the derivation; a Contraction can be swapped upwards into any rule other than an Application or Product, and can also be swapped upwards into the relevant side of an Application or Product where both variables being contracted come from the same side. After that, any remaining Contractions are replaced via identifying the two variables being contracted inside the Application or Product that is being contracted (this violates the $\Gamma \# \Delta$ constraint of explicitly contracting ICA, but this is not a problem because that is legal in implicitly contracting ICA). Then Applications and Products are replaced; let $x_1 \dots x_j$ be the free variables of the term derived by ∇_1 that don't appear in the term derived by ∇_2, and $y_1 \dots y_k$ be the free variables of the term derived by ∇_2 that don't appear in the term derived by ∇_1:

$$\begin{aligned}
App(\nabla_1,\nabla_2) &\mapsto App(Weak(y_1,Weak(\ldots,Weak(y_k,\nabla_1)\ldots)),\\
&\qquad Weak(x_1,Weak(\ldots,Weak(x_j,\nabla_2)\ldots)))\\
Prod(\nabla_1,\nabla_2) &\mapsto Prod(Weak(y_1,Weak(\ldots,Weak(y_k,\nabla_1)\ldots)),\\
&\qquad Weak(x_1,Weak(\ldots,Weak(x_j,\nabla_2)\ldots)))
\end{aligned}$$

In other words, Applications and Products that don't have the same free variables in their contexts are made to have the same contexts via adding extra Weakenings. Again, there must be extra uses of Exchange, not shown in the above replacements, to move the variables into the correct order for the Weakenings, and while moving the Contractions. $\square$

In this explicitly contracting version of ICA, the Weakening rule is used only when a function ignores its argument (i.e. uses its argument less than once), and the Contraction rule is used when a function uses its argument more than once. We could produce a *linear* type system, a type system in which each function uses its argument exactly once, by deleting both rules (an approach that is not entirely unprecedented; such a language, Rudimentary Linear Logic, was used by Girard et al as an example in [23, page 7]). In this case, however, we care about an argument being used more than once, but not about an argument being used less than once, and so we delete merely the Contraction rule, to create an *affine* type system, Affine ICA. The change in semantics also comes with two changes in notation. Because sharing is no longer allowed between the two halves of a "product", it is actually a tensor, rather than a product, and so we change our notation to match. (We take $M_1 \otimes (M_2 \otimes M_3)$ and $(M_1 \otimes M_2) \otimes M_3$ to be syntactically different terms, though, just as $\langle M_1, \langle M_2, M_3\rangle\rangle$ is a syntactically different term from $\langle\langle M_1, M_2\rangle, M_3\rangle$; the choice of whether tuple notation is associative or not is mostly arbitrary, so we make the same choice as in ICA itself.) Likewise, in the absence of contraction, all functions are affine (i.e. use their argument at most once); as such, we use the notation $\theta' \multimap \theta$,

$$\theta ::= \mathsf{exp}_J \mid \theta \multimap \theta \mid \theta \otimes \theta, \text{ where } J \subset \mathbb{N} \text{ is a finite set}$$
$$\Gamma ::= \text{sequence of } x\!:\!\theta$$
$$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$

$$\frac{x \neq y}{x\!:\!\theta \# y\!:\!\theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma :: \Gamma'\#\Delta} \qquad \frac{x\!:\!\theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{x\!:\!\theta \vdash x\!:\!\theta}\text{ Identity}$$

$$\frac{M\!:\!\theta \text{ is a constant}}{\vdash M\!:\!\theta}\text{ Constant}$$

$$\frac{\Gamma \vdash M\!:\!\theta \qquad x\#\Gamma}{\Gamma :: x\!:\!\theta' \vdash M\!:\!\theta}\text{ Weakening}$$

$$\frac{\Gamma :: x\!:\!\theta' :: y\!:\!\theta'' :: \Delta \vdash M\!:\!\theta}{\Gamma :: y\!:\!\theta'' :: x\!:\!\theta' :: \Delta \vdash M\!:\!\theta}\text{ Exchange}$$

$$\frac{\Gamma :: x\!:\!\theta' :: \Delta \vdash M\!:\!\theta}{\Gamma :: \Delta \vdash \lambda(x\!:\!\theta').M\!:\!(\theta' \multimap \theta)}\text{ Abstraction}$$

$$\frac{\Gamma \vdash M\!:\!(\theta' \multimap \theta) \qquad \Delta \vdash N\!:\!\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash MN\!:\!\theta}\text{ Application}$$

$$\frac{\Gamma \vdash M\!:\!\theta \qquad \Delta \vdash N\!:\!\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash M \otimes N\!:\!(\theta \otimes \theta')}\text{ Tensor}$$

Figure 5.1.3: Affine ICA

rather than $\theta' \to \theta$, as a reminder of this fact (this notation comes from linear logic). We also make one other change, to avoid an unbounded amount of state being introduced via the use of unbounded integers: instead of an exp type that ranges over all integers (and thus has infinitely many values), we parameterize our exp type to have only finitely many possible values. (This also means that we do not need a separate com type, because com is equivalent to $\mathsf{exp}_{\{0\}}$, the type of expressions that can only evaluate to a single result. And we do not need a separate var or sem type because there are no constants that use them; the definition that we will use for newvar handles reading the variable and writing the variable separately, as terms of form exp and $\mathsf{exp} \multimap \mathsf{com}$ respectively, rather than via using a var type.) Our new type system, defining a new set of terms, is shown in Figure 5.1.3 (again omitting constants).

In order to ensure that the type system remains truly affine, we also need to restrict the constants we use. Constants like newsem are pointless in an affine type system, because it would be impossible for two terms to access the same semaphore simultaneously and thus be able to synchronize on it. We also need to remove fix_θ, because it uses its argument more than once (both as an argument, and as a function); and while, because it also uses its arguments more than once (the condition is evaluated between each loop iteration, and the loop body is evaluated each time through the loop). Interesting constants that we can continue to use with Affine ICA include parallel and sequential composition (par and seq), and conditional execution (if), in addition to projections, arithmetic operations and base type constants like skip.

Perhaps surprisingly, though, we do not take the projections as constants of Affine ICA. The reason is down to the difference between tensors and products. In a product type $\theta \times \theta'$, it is possible for the two halves of the product to share; the same free variables can exist on both sides of the product. (This is why ICA and bSCI both use product notation; see Chapter 8 for much more explanation on this point, and Section 3.2 for a discussion of the difference between tensors and products.) Thus, in an affine type system, we have $\pi_1 : (\theta \times \theta') \multimap \theta$ and $\pi_2 : (\theta \times \theta') \multimap \theta'$; we have to choose which side of the product we want to look at, because looking at both at once would violate the affine nature of the type system due to the potential for sharing. However, in a tensor type $\theta \otimes \theta'$, the two halves of the tensor cannot share, which is the situation that we have in Affine ICA. We can see this both from the type system (which disallows any sort of contraction between the two halves of a tensor, just as it disallows it anywhere else), and for categorical reasons: with a typical categorical semantics, $(\theta'_1 \otimes \theta'_2) \multimap \theta$ and $\theta'_1 \multimap (\theta'_2 \multimap \theta)$ are isomorphic types (because their denotations are $(\llbracket\theta'_1\rrbracket \otimes \llbracket\theta'_2\rrbracket) \Rightarrow \llbracket\theta\rrbracket$ and $\llbracket\theta'_1\rrbracket \Rightarrow (\llbracket\theta'_2\rrbracket \Rightarrow \llbracket\theta\rrbracket)$ which are isomorphic in a closed category), and it is clear that the two halves of the tensor cannot share in the second of these terms. We could define $\pi_1 : (\theta \otimes \theta') \multimap \theta$ and $\pi_2 : (\theta \otimes \theta') \multimap \theta'$, analogously to the definitions for products, but these constants would be insufficient for reading both sides of a given tensor, as they each discard half of a tensor as they retrieve the other half; we would prefer a constant capable of accessing both halves of a tensor, rather than just one. The simplest such constant is $\mathsf{uncurry}_{\theta'_1,\theta'_2,\theta} : (\theta'_1 \multimap (\theta'_2 \multimap \theta)) \multimap ((\theta'_1 \otimes$

$$
\begin{aligned}
\mathsf{skip} &: \mathsf{com}\\
j_J &: \mathsf{exp}_J\\
\mathsf{seq}_J &: \mathsf{com} \multimap (\mathsf{exp}_J \multimap \mathsf{exp}_J)\\
\mathsf{par} &: \mathsf{com} \multimap (\mathsf{com} \multimap \mathsf{com})\\
\mathsf{if}_J &: \mathsf{exp}_{\{0,1\}} \multimap (\mathsf{exp}_J \multimap (\mathsf{exp}_J \multimap \mathsf{exp}_J))\\
\mathsf{uncurry}_{\theta'_1,\theta'_2,\theta} &: (\theta'_1 \multimap (\theta'_2 \multimap \theta)) \multimap ((\theta'_1 \otimes \theta'_2) \multimap \theta)\\
\mathsf{op}_{J,\bullet} &: \mathsf{exp}_J \multimap (\mathsf{exp}_J \multimap \mathsf{exp}_J)\\
\mathsf{newvar}_{J,r,w} &: ((((\overbrace{\mathsf{exp}_J \otimes (\ldots \otimes \mathsf{exp}_J)}^{r \text{ times}}) \otimes (\overbrace{(\mathsf{exp}_J \multimap \mathsf{com}) \otimes (\ldots \otimes (\mathsf{exp}_J \multimap \mathsf{com})}^{w \text{ times}})))\\
&\quad \multimap \mathsf{com}) \multimap \mathsf{com})
\end{aligned}
$$

Figure 5.1.4: Constants of Affine ICA

$\theta'_2) \multimap \theta)$, which can be thought of as taking a function which takes two separate arguments and transforming it into a function that takes both arguments as a single tensor. And with uncurry, we no longer need the projections as constants, because they can be defined as terms instead: $\pi_{1,\theta,\theta'} \triangleq \mathsf{uncurry}_{\theta,\theta',\theta}(\lambda(x\!:\!\theta).\lambda(y\!:\!\theta').x)$ and $\pi_{2,\theta,\theta'} \triangleq \mathsf{uncurry}_{\theta,\theta',\theta'}(\lambda(x\!:\!\theta).\lambda(y\!:\!\theta').y)$.

There is one other constant that can be used, but needs changing. For newvar, the definition from ICA is not very useful (the variable could only be written once, and read once), but it's possible to imagine a family of newvar constants that give values other than 1 for the number of times that the variable can be read and written. As such, we can replace ICA's newvar with a version designed for Affine ICA that has separate read and write arguments: $\mathsf{newvar}_{J,r,w} : (((\overbrace{\mathsf{exp}_J \otimes (\ldots \otimes \mathsf{exp}_J)}^{r \text{ times}}) \otimes \overbrace{(\mathsf{exp}_J \multimap \mathsf{com}) \otimes (\ldots \otimes (\mathsf{exp}_J \multimap \mathsf{com}))}^{w \text{ times}}) \multimap \mathsf{com}) \multimap \mathsf{com})$ which provides r different arguments to read the variable via, and w different arguments to write the variable via. In ICA, the newvar of Affine ICA can be defined as $\lambda f.\mathsf{newvar}(\lambda x.f(\mathsf{deref}(x) \times \ldots \times \mathsf{deref}(x) \times \mathsf{assign}(x) \times \ldots \times \mathsf{assign}(x)))$; this definition does not work in Affine ICA due to the multiple uses of x, which is why a separate constant is needed.

The constants of Affine ICA are listed in Figure 5.1.4, for convenient reference.

5.2 Categorical semantics of Affine ICA

In order to think about the semantics of Affine ICA, we define a categorical semantics – basically, a framework that we would expect other semantics to fit into. More precisely, we define a set of axioms such that we'd expect any semantics of Affine ICA to be consistent with those axioms. The semantics itself is shown in equation form in Figure 5.2.1 and diagram form in Figure 5.2.2; what follows is a description of why the particular choices in that semantics were made. (To explain the notation: a natural transformation from F to G, with F and G functors from $\mathscr{D}$ to $\mathscr{E}$, is a function from objects A of $\mathscr{D}$ to morphisms on $F(A) \longrightarrow G(A)$ of $\mathscr{E}$. Thus, the "type" of a natural transformation is expressed as $F \dot{\rightarrow} G$ (or $F \cong G$ for a natural isomorphism), where F and G are functors. The functors in turn are expressed in the form $(A \mapsto B, f \mapsto g)$; this notation shows the effect of the functor on objects and on morphisms. Because many of the functors are from product categories, the notation can get quite complex.)

As is usual for a categorical semantics, we define a category $\mathscr{C}$ such that the denotations of types are objects of $\mathscr{C}$, and the denotations of terms are morphisms of $\mathscr{C}$. In order to have the correct types, we need the category to contain denotations for function types, $[\![\theta \multimap \theta']\!] = [\![\theta]\!] \Rightarrow [\![\theta']\!]$, and denotations for tensor types, $[\![\theta \otimes \theta']\!] = [\![\theta]\!] \otimes [\![\theta']\!]$. (The category will also need a denotation $[\![\mathsf{exp}_J]\!]$, but at this level of generality, there is nothing we can say about it other than that it exists.) As such, the appropriate structure for the category is a monoidal closed category (this is standard for categories with functions and tensors); this structure is defined in Figure 5.2.1. We also need to be able to denote both closed and open judgements, which means that we need to decide which objects they are morphisms from and to. The denotation for a closed judgement is simple enough; $[\![\vdash M : \theta]\!]$ is a morphism from I (the unit of $\mathscr{C}$'s tensor) to $[\![\theta]\!]$ (i.e. $[\![\vdash M : \theta]\!] : I \longrightarrow [\![\theta]\!]$). For open terms (which have open judgements), the situation is more complex, due to two considerations that cause problems in combination. One problem is that $x : \mathsf{com} :: y : \mathsf{com} \vdash \mathsf{seq}(x)(y)$ is not the same term as $x : \mathsf{com} :: y : \mathsf{com} \vdash \mathsf{seq}(y)(x)$; i.e. in an open term, the variable names matter. The other problem is we need two terms which are observationally equivalent to have identical denotations. We solve this problem via the standard

$\mathscr{C}$ is monoidal (axiomatization by Kelly and MacLane in [27]):

$$\begin{aligned}
\otimes : \mathscr{C} \times \mathscr{C} \to \mathscr{C} &\text{ is a functor} \\
I &\text{ is an object of } \mathscr{C} \\
\rho_A : (A \mapsto A \otimes I, f \mapsto f \otimes \mathbf{id}_I) \cong (A \mapsto A, f \mapsto f) &\text{ is a natural isomorphism} \\
\alpha_{A,B,C} : ((A,B,C \mapsto (A \otimes B) \otimes C), & \\
(f,g,h \mapsto (f \otimes g) \otimes h)) & \\
\cong ((A,B,C \mapsto A \otimes (B \otimes C)), & \\
(f,g,h \mapsto f \otimes (g \otimes h))) &\text{ is a natural isomorphism} \\
\alpha_{A\otimes B,C,D}; \alpha_{A,B,C\otimes D} &= (\alpha_{A,B,C} \otimes \mathbf{id}_D); \alpha_{A,B\otimes C,D}; (\mathbf{id}_A \otimes \alpha_{B,C,D}) \\
\alpha_{A,B,I}; (\mathbf{id}_A \otimes \rho_B) &= \rho_{A\otimes B}
\end{aligned}$$

$\mathscr{C}$ is symmetric (axiomatization by Kelly and MacLane in [27]):

$$\begin{aligned}
\gamma_{A,B} : ((A,B \mapsto A \otimes B), (f,g \mapsto f \otimes g)) & \\
\cong ((A,B \mapsto B \otimes A), (f,g \mapsto g \otimes f)) &\text{ is a natural isomorphism} \\
\gamma_{A,B}; \gamma_{B,A} &= \mathbf{id}_{A\otimes B} \\
\alpha_{A,B,C}; \gamma_{A,B\otimes C}; \alpha_{B,C,A} &= (\gamma_{A,B} \otimes \mathbf{id}_C); \alpha_{B,A,C}; (\mathbf{id}_B \otimes \gamma_{A,C})
\end{aligned}$$

$\mathscr{C}$ is closed (axiomatization by Kelly and MacLane in [27]):

$$\begin{aligned}
\Rightarrow : \mathscr{C}^{op} \times \mathscr{C} \to \mathscr{C} &\text{ is a functor} \\
\mathbf{abs}_{A,B} : ((A,B \mapsto A), (f,g \mapsto f)) & \\
\to ((A,B \mapsto B \Rightarrow (A \otimes B)), & \\
(f,g \mapsto g^{op} \Rightarrow (f \otimes g))) &\text{ is a natural transformation} \\
\mathbf{eval}_{A,B} : ((A,B \mapsto (A \Rightarrow B) \otimes A), & \\
(f,g \mapsto (f^{op} \Rightarrow g) \otimes f)) & \\
\to ((A,B \mapsto B), (f,g \mapsto g)) &\text{ is a natural transformation} \\
\mathbf{abs}_{A\Rightarrow B,A}; (\mathbf{id}_A \Rightarrow \mathbf{eval}_{A,B}) &= \mathbf{id}_{A\Rightarrow B} \\
(\mathbf{abs}_{A,B} \otimes \mathbf{id}_B); \mathbf{eval}_{B,A\otimes B} &= \mathbf{id}_{A\otimes B}
\end{aligned}$$

$\mathscr{C}$ is affine (axiomatization by Petrić in [37]):

$$\begin{aligned}
\top_A : A \longrightarrow I &\text{ is a family of morphisms} \\
\top_I &= \mathbf{id}_I \\
\forall f : A \longrightarrow B. \top_A &= f; \top_B
\end{aligned}$$

Figure 5.2.1: Categorical structure of Affine ICA's categorical semantics

$$\begin{tikzcd}
((A \otimes B) \otimes C) \otimes D \arrow[r, "\alpha_{A,B,C} \otimes \mathbf{id}_D"] \arrow[d, "\alpha_{A \otimes B,C,D}"'] & (A \otimes (B \otimes C)) \otimes D \arrow[dd, "\alpha_{A,B \otimes C,D}"] \\
(A \otimes B) \otimes (C \otimes D) \arrow[d, "\alpha_{A,B,C \otimes D}"'] & \\
A \otimes (B \otimes (C \otimes D)) & A \otimes ((B \otimes C) \otimes D) \arrow[l, "\mathbf{id}_A \otimes \alpha_{B,C,D}"]
\end{tikzcd}$$

$$\begin{tikzcd}
(A \otimes B) \otimes C \arrow[r, "\gamma_{A,B} \otimes \mathbf{id}_C"] \arrow[d, "\alpha_{A,B,C}"'] & (B \otimes A) \otimes C \arrow[d, "\alpha_{B,A,C}"] \\
A \otimes (B \otimes C) \arrow[d, "\gamma_{A,B \otimes C}"'] & B \otimes (A \otimes C) \arrow[d, "\mathbf{id}_B \otimes \gamma_{A,C}"] \\
(B \otimes C) \otimes A \arrow[r, "\alpha_{B,C,A}"'] & B \otimes (C \otimes A)
\end{tikzcd}$$

$$\begin{tikzcd}
A \otimes B \arrow[d, shift left, "\gamma_{A,B}"] \\
B \otimes A \arrow[u, shift left, "\gamma_{B,A}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
(A \otimes B) \otimes I \arrow[rr, "\alpha_{A,B,I}"] \arrow[dr, "\rho_{A \otimes B}"'] & & A \otimes (B \otimes I) \arrow[dl, "\mathbf{id}_A \otimes \rho_B"] \\
& A \otimes B &
\end{tikzcd}
\qquad
\begin{tikzcd}
I \arrow[d, shift right, "\mathbf{id}_I"'] \arrow[d, shift left, "\top_I"] \\
I
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[rr, "f"] \arrow[dr, "\top_A"'] & & B \arrow[dl, "\top_B"] \\
& I &
\end{tikzcd}$$

$$\begin{tikzcd}
& A \Rightarrow ((A \Rightarrow B) \otimes A) \arrow[d, "\mathbf{id}_A \Rightarrow \mathbf{eval}_{A,B}"] \\
A \Rightarrow B \arrow[ur, "\mathbf{abs}_{A \Rightarrow B,A}"] \arrow[r, "\mathbf{id}_{A \Rightarrow B}"'] & A \Rightarrow B
\end{tikzcd}
\qquad
\begin{tikzcd}
& (B \Rightarrow (A \otimes B)) \otimes B \arrow[d, "\mathbf{eval}_{B,A \otimes B}"] \\
A \otimes B \arrow[ur, "(\mathbf{abs}_{A,B}) \otimes \mathbf{id}_B"] \arrow[r, "\mathbf{id}_{A \otimes B}"'] & A \otimes B
\end{tikzcd}$$

Figure 5.2.2: Equations in Figure 5.2.1, as commutative diagrams

method of writing denotations for judgements, not closed terms, and assigning an arbitrary order to free variables in the judgement; this is the reason that we use sequences of $x : \theta$, rather than sets (which would be more natural). This then allows us to define the endpoints of a morphism that denotes an open judgement; we want something like $[\![x_1 : \theta_1 :: x_2 : \theta_2 :: \ldots :: x_j : \theta_j \vdash M : \theta]\!] : ([\![\theta_1]\!] \otimes [\![\theta_2]\!] \otimes \ldots \otimes [\![\theta_j]\!]) \longrightarrow [\![\theta]\!]$.

We need to exercise some care here. Sequence concatenation is a strictly associative operation, so $x_1 : \theta_1 :: x_2 : \theta_2 :: x_3 : \theta_3$ produces the same result regardless of which concatenation is done first. However, $(A \otimes B) \otimes C$ and $A \otimes (B \otimes C)$ are not necessarily equal; they must be isomorphic (by the definition of a monoidal category), but the fact that they may be different means that some concrete associativity must be chosen for the denotation $[\![\theta_1]\!] \otimes [\![\theta_2]\!] \otimes \ldots \otimes [\![\theta_j]\!]$ of a context $x_1 : \theta_1 :: x_2 : \theta_2 :: \ldots :: x_j : \theta_j$. We arbitrarily choose the left-associative definition, $[\![x_1 : \theta_1 :: x_2 : \theta_2 :: \ldots :: x_j : \theta_j \vdash M : \theta]\!] : ((((I \otimes [\![\theta_1]\!]) \otimes [\![\theta_2]\!]) \otimes \ldots) \otimes [\![\theta_j]\!]) \longrightarrow [\![\theta]\!]$. Note that an extra I has been added at the innermost point of the tensor; this is to handle the special case of $[\![\varepsilon]\!]$ in a way consistent with other contexts, thus avoiding needing a special case for it in every

We define natural isomorphisms α^*, γ^* and a transformation $\mathbf{abs}^*$ natural in Γ, Δ, θ:

$$\begin{aligned}
\alpha^*_{\Gamma,\Delta} &= \llbracket\Gamma :: \Delta\rrbracket \longrightarrow \llbracket\Gamma\rrbracket \otimes \llbracket\Delta\rrbracket \\
\alpha^*_{\Gamma,\varepsilon} &= \rho^{-1}_{\llbracket\Gamma\rrbracket} \\
\alpha^*_{\Gamma,\Delta::x:\theta} &= (\alpha^*_{\Gamma,\Delta} \otimes \mathbf{id}_{\llbracket\theta\rrbracket});\, \alpha_{\llbracket\Gamma\rrbracket,\llbracket\Delta\rrbracket,\llbracket\theta\rrbracket} \\
\gamma^*_{\Gamma,x:\theta,y:\theta',\Delta} &: \llbracket\Gamma :: x:\theta :: y:\theta' :: \Delta\rrbracket \longrightarrow \llbracket\Gamma :: y:\theta' :: x:\theta :: \Delta\rrbracket \\
\gamma^*_{\Gamma,x:\theta,y:\theta',\Delta} &= \alpha^*_{\Gamma::x:\theta::y:\theta',\Delta}; \\
&\quad ((\alpha_{\llbracket\Gamma\rrbracket,\llbracket\theta\rrbracket,\llbracket\theta'\rrbracket};\, (\mathbf{id}_{\llbracket\Gamma\rrbracket} \otimes \gamma_{\llbracket\theta\rrbracket,\llbracket\theta'\rrbracket});\, \alpha^{-1}_{\llbracket\Gamma\rrbracket,\llbracket\theta'\rrbracket,\llbracket\theta\rrbracket}) \otimes \mathbf{id}_{\llbracket\Delta\rrbracket}); \\
&\quad \alpha^{*-1}_{\Gamma::x:\theta::y:\theta',\Delta} \\
\mathbf{abs}^*_{\Gamma,x:\theta',\Delta,\theta} &: (\llbracket\Gamma :: x : \theta' :: \Delta\rrbracket \longrightarrow \llbracket\theta\rrbracket) \to (\llbracket\Gamma :: \Delta\rrbracket \longrightarrow \llbracket\theta' \Rightarrow \theta\rrbracket) \\
\mathbf{abs}^*_{\Gamma,x:\theta',\Delta,\theta}(f) &= \alpha^*_{\Gamma,\Delta};\, \gamma_{\llbracket\Gamma\rrbracket,\llbracket\Delta\rrbracket};\, \mathbf{abs}_{\llbracket\Delta\rrbracket\otimes\llbracket\Gamma\rrbracket,\llbracket\theta'\rrbracket}; \\
&\quad (\mathbf{id}_{\llbracket\theta'\rrbracket} \Rightarrow (\alpha_{\llbracket\Delta\rrbracket,\llbracket\Gamma\rrbracket,\llbracket\theta'\rrbracket};\, \gamma_{\llbracket\Delta\rrbracket,\llbracket\Gamma\rrbracket\otimes\llbracket\theta'\rrbracket};\, \alpha^{*-1}_{\Gamma::x:\theta',\Delta};\, f))
\end{aligned}$$

Denotations of types:

$$\begin{aligned}
\llbracket\theta' \multimap \theta\rrbracket &= \llbracket\theta'\rrbracket \Rightarrow \llbracket\theta\rrbracket \\
\llbracket\theta \otimes \theta'\rrbracket &= \llbracket\theta\rrbracket \otimes \llbracket\theta'\rrbracket
\end{aligned}$$

Denotations of contexts:

$$\begin{aligned}
\llbracket\varepsilon\rrbracket &= I \\
\llbracket\Gamma :: x:\theta\rrbracket &= \llbracket\Gamma\rrbracket \otimes \llbracket\theta\rrbracket
\end{aligned}$$

Denotations of derivations:

$$\begin{aligned}
\llbracket\Gamma \vdash M:\theta\rrbracket &: \llbracket\Gamma\rrbracket \longrightarrow \llbracket\theta\rrbracket \\
\llbracket Id_{x:\theta} : (x:\theta \vdash x:\theta)\rrbracket &= \gamma_{I,\llbracket\theta\rrbracket};\, \rho_{\llbracket\theta\rrbracket} \\
\llbracket Weak_{x:\theta'}(\nabla) : (\Gamma :: x:\theta' \vdash M:\theta)\rrbracket &= (\mathbf{id}_{\llbracket\Gamma\rrbracket} \otimes \top_{\llbracket\theta'\rrbracket});\, \rho_{\llbracket\Gamma\rrbracket};\, \llbracket\nabla : (\Gamma \vdash M:\theta)\rrbracket \\
\llbracket Abs_x(\nabla) : (\Gamma :: \Delta \vdash \lambda(x:\theta').M : (\theta' \multimap \theta))\rrbracket &= \mathbf{abs}^*_{\Gamma,x:\theta',\Delta,\theta}(\llbracket\nabla : (\Gamma :: x:\theta' :: \Delta \vdash M:\theta)\rrbracket) \\
\llbracket App(\nabla,\nabla') : (\Gamma :: \Delta \vdash MN:\theta)\rrbracket &= \alpha^*_{\Gamma,\Delta};\, (\llbracket\nabla : (\Gamma \vdash M : (\theta' \multimap \theta))\rrbracket \otimes \\
&\qquad\qquad \llbracket\nabla' : (\Delta \vdash N:\theta')\rrbracket);\mathbf{eval}_{\llbracket\theta'\rrbracket,\llbracket\theta\rrbracket} \\
\llbracket Tens(\nabla,\nabla') : (\Gamma :: \Delta \vdash M \otimes N : (\theta \otimes \theta'))\rrbracket &= \alpha^*_{\Gamma,\Delta};\, (\llbracket\nabla : (\Gamma \vdash M:\theta)\rrbracket \otimes \llbracket\nabla' : (\Delta \vdash N:\theta')\rrbracket) \\
\llbracket Exch_{y,x}(\nabla) : (\Gamma :: x:\theta' :: y:\theta'' :: \Delta \vdash M:\theta)\rrbracket &= \gamma^*_{\Gamma,x:\theta',y:\theta'',\Delta};\, \llbracket\nabla : (\Gamma :: y:\theta'' :: x:\theta' :: \Delta \vdash M:\theta)\rrbracket
\end{aligned}$$

Figure 5.2.3: Denotations of terms in Affine ICA's categorical semantics

$$\begin{tikzcd}
\llbracket \Gamma :: x\colon\theta :: y\colon\theta' :: \Delta \rrbracket \arrow[r, "\gamma^*_{\Gamma,x:\theta,y:\theta',\Delta}"] \arrow[d, "\alpha^*_{\Gamma::x:\theta::y:\theta',\Delta}"'] & \llbracket \Gamma :: y\colon\theta' :: x\colon\theta :: \Delta \rrbracket \\
\llbracket \Gamma :: x\colon\theta :: y\colon\theta' \rrbracket \otimes \llbracket \Delta \rrbracket \arrow[d, "\alpha_{\llbracket\Gamma\rrbracket,\llbracket\theta\rrbracket,\llbracket\theta'\rrbracket}\otimes\mathbf{id}_{\llbracket\Delta\rrbracket}"'] & \llbracket \Gamma :: y\colon\theta' :: x\colon\theta \rrbracket \otimes \llbracket \Delta \rrbracket \arrow[u, "\alpha^{*-1}_{\Gamma::x:\theta::y:\theta',\Delta}"'] \\
\llbracket \Gamma \rrbracket \otimes (\llbracket \theta \rrbracket \otimes \llbracket \theta' \rrbracket) \otimes \llbracket \Delta \rrbracket \arrow[r, "(\mathbf{id}_{\llbracket\Gamma\rrbracket}\otimes\gamma_{\llbracket\theta\rrbracket,\llbracket\theta'\rrbracket})\otimes\mathbf{id}_{\llbracket\Delta\rrbracket}"'] & \llbracket \Gamma \rrbracket \otimes (\llbracket \theta' \rrbracket \otimes \llbracket \theta \rrbracket) \otimes \llbracket \Delta \rrbracket \arrow[u, "\alpha^{-1}_{\llbracket\Gamma\rrbracket,\llbracket\theta'\rrbracket,\llbracket\theta\rrbracket}\otimes\mathbf{id}_{\llbracket\Delta\rrbracket}"']
\end{tikzcd}$$

$$\begin{tikzcd}
\llbracket \Gamma :: \Delta \rrbracket \arrow[r, "\mathbf{abs}^*_{\Gamma,x:\theta',\Delta,\theta}(f)"] \arrow[d, "\alpha^*_{\Gamma,\Delta}"'] & \llbracket \theta' \Rightarrow \theta \rrbracket \\
\llbracket \Gamma \rrbracket \otimes \llbracket \Delta \rrbracket \arrow[d, "\gamma_{\llbracket\Gamma\rrbracket,\llbracket\Delta\rrbracket}"'] & \llbracket \theta' \rrbracket \Rightarrow \llbracket \Gamma :: x\colon\theta' :: \Delta \rrbracket \arrow[u, "\mathbf{id}_{\llbracket\theta'\rrbracket}\Rightarrow f"'] \\
\llbracket \Delta \rrbracket \otimes \llbracket \Gamma \rrbracket \arrow[d, "\mathbf{abs}_{\llbracket\Delta\rrbracket\otimes\llbracket\Gamma\rrbracket,\llbracket\theta'\rrbracket}"'] & \llbracket \theta' \rrbracket \Rightarrow (\llbracket \Gamma :: x\colon\theta' \rrbracket \otimes \llbracket \Delta \rrbracket) \arrow[u, "\mathbf{id}_{\llbracket\theta'\rrbracket}\Rightarrow\alpha^{*-1}_{\Gamma::x:\theta',\Delta}"'] \\
\llbracket \theta' \rrbracket \Rightarrow ((\llbracket \Delta \rrbracket \otimes \llbracket \Gamma \rrbracket) \otimes \llbracket \theta' \rrbracket) \arrow[r, "\mathbf{id}_{\llbracket\theta'\rrbracket}\Rightarrow\alpha_{\llbracket\Delta\rrbracket,\llbracket\Gamma\rrbracket,\llbracket\theta'\rrbracket}"'] & \llbracket \theta' \rrbracket \Rightarrow (\llbracket \Delta \rrbracket \otimes \llbracket \Gamma :: x\colon\theta' \rrbracket) \arrow[u, "\mathbf{id}_{\llbracket\theta'\rrbracket}\Rightarrow\gamma_{\llbracket\Delta\rrbracket,\llbracket\Gamma\rrbracket\otimes\llbracket\theta'\rrbracket}"']
\end{tikzcd}$$

Figure 5.2.4: Definitions of γ^* and $\mathbf{abs}^*$ drawn as diagrams

rule.

Some specific rules also end up requiring specific properties of the category. In order to be able to denote the result of a Weakening, we need to be able to find a unique morphism that allows us to discard the part of our free variable tensor that corresponds to the variable that's being weakened. As such, the category needs a terminal object; and because we want to be able to eliminate that object from the free variable tensor afterwards, the category must be affine (i.e. the terminal object needs to be the unit of the tensor). And in order to be able to implement Exchange, we need to be able to exchange elements of tensors too; the category must be at least braided (i.e. some γ exists that exchanges the sides of a tensor) for this operation to even be definable, and symmetric (i.e. γ is self-inverse) for it to be coherent.

We can now define the categorical semantics inductively on a derivation tree. The semantics is shown in Figure 5.2.3; to make it easier to follow, we give "types" for the derivations, using the notation $\nabla : (\Gamma \vdash M : \theta)$ to mean "a derivation tree ∇ that derives the judgement $\Gamma \vdash M :$

θ". The main result that needs to be proved about this categorical semantics is coherence; the definition in that figure defines the semantics of a derivation $[\![\nabla]\!]$, but what we really want is a denotation of a judgement. Due to the possibility of changing the free variable order (or adding new variables via weakening) at arbitrary points when deriving the denotation, it is not obvious that the denotation of a term is identical no matter which rules of the type system are applied in order to derive it.

Lemma 5.2.1. *Given two Affine ICA derivation trees* $\nabla : (\Gamma \vdash M : \theta)$, $\nabla' : (\Gamma \vdash M : \theta)$ *for which each subtree* $\nabla'' : (\Gamma'' \vdash M'' : \theta'')$ *that is rooted at the premise of an Abstraction has* $\Gamma'' = \mathbf{fv}(\nabla'')$ *(with the definition of* $\mathbf{fv}$ *from Figure 5.2.5), then* $[\![\nabla]\!] = [\![\nabla']\!]$.

Proof. We define morphisms $[\![\nabla]\!]_L$, $[\![\nabla']\!]_R$ using the definitions in Figure 5.2.5. We observe that by structural induction on ∇, $[\![\nabla]\!]_L ; [\![\nabla]\!]_R = [\![\nabla]\!]$, and likewise for ∇'. $\mathbf{fv}(\nabla) = \mathbf{fv}(\nabla')$ and $[\![\nabla]\!]_R = [\![\nabla']\!]_R$ by structural induction on M; Weakening and Exchange have no effect on $\mathbf{fv}$ or $[\![-]\!]_R$, the effect of Abstraction is fixed by the fact that its premise has (by the same structural induction) an $\mathbf{fv}$ and thus (by assumption) a context that depends only on M, and when and whether to use the other rules (Identity, Constant, Application and Tensor) is likewise fixed via the syntax of M.

It is an existing result (proved by Petrić in [37, Theorem 2 on page 14]) that in an affine symmetric monoidal category, that two natural transformations between the same functors, each built solely out of α, α^{-1}, γ, ρ, ρ^{-1}, $\top$, $\mathbf{id}$, $\otimes$, and morphism composition, are equal. This immediately proves that $[\![\nabla]\!]_L = [\![\nabla']\!]_L$; thus $[\![\nabla]\!] = [\![\nabla]\!]_L ; [\![\nabla]\!]_R = [\![\nabla']\!]_L ; [\![\nabla']\!]_R = [\![\nabla']\!]$. □

Lemma 5.2.2. *Given an Affine ICA derivation tree* $\nabla : (\Gamma \vdash M : \theta)$, *there is a derivation tree* ∇_c *such that* $[\![\nabla]\!] = [\![\nabla_c]\!]$, *and such that for each subtree* $\nabla'' : (\Gamma'' \vdash M'' : \theta'')$ *in* ∇_c *that is rooted at the premise of an Abstraction,* $\Gamma'' = \mathbf{fv}(\nabla'')$.

Proof. Proof is by structural induction on ∇. In the base case, where M is a variable or constant, $\nabla_c = \nabla$ fulfills the required condition. For all the inductive cases where the last rule applied in

$$
\begin{aligned}
\mathbf{fv}(Id_{x:\theta} : (x{:}\,\theta \vdash x{:}\,\theta)) &= x : \theta \\
\mathbf{fv}(Const_M : (\vdash M{:}\,\theta)) &= \varepsilon \\
\mathbf{fv}(App(\nabla,\nabla') : (\Gamma :: \Delta \vdash MN{:}\,\theta)) &= \mathbf{fv}(\nabla : (\Gamma \vdash M{:}\,(\theta' \multimap \theta))) :: \mathbf{fv}(\Delta \vdash N{:}\,\theta') \\
\mathbf{fv}(Tens(\nabla,\nabla') : (\Gamma :: \Delta \vdash M \otimes N{:}\,(\theta \otimes \theta'))) &= \mathbf{fv}(\nabla : (\Gamma \vdash M{:}\,\theta)) :: \mathbf{fv}(\nabla' : (\Delta \vdash N{:}\,\theta')) \\
\mathbf{fv}(Weak_{x:\theta'}(\nabla) : (\Gamma :: x{:}\,\theta' \vdash M{:}\,\theta)) &= \mathbf{fv}(\nabla : (\Gamma \vdash M{:}\,\theta)) \\
\mathbf{fv}(Exch_{y,x}(\nabla) : (\Gamma :: x{:}\,\theta' :: y{:}\,\theta'' :: \Delta \vdash M{:}\,\theta)) &= \mathbf{fv}(\nabla : (\Gamma :: y{:}\,\theta' :: x{:}\,\theta'' :: \Delta \vdash M{:}\,\theta)) \\
\mathbf{fv}(Abs_x(\nabla) : (\Gamma :: \Delta \vdash \lambda(x{:}\,\theta').M{:}\,(\theta' \multimap \theta))) &= \Gamma :: \Delta
\end{aligned}
$$

$$
\begin{aligned}
[\![\nabla : (\Gamma \vdash M{:}\,\theta)]\!]_L &: [\![\Gamma]\!] \longrightarrow [\![\mathbf{fv}(\nabla)]\!] \\
[\![Id_{x:\theta} : (x{:}\,\theta \vdash x{:}\,\theta)]\!]_L &= \gamma_{I,[\![\theta]\!]}; \rho_{[\![\theta]\!]} \\
[\![Const_M : (\vdash M{:}\,\theta)]\!]_L &= \mathbf{id}_{[\![\theta]\!]} \\
[\![App(\nabla,\nabla') : (\Gamma :: \Delta \vdash MN{:}\,\theta)]\!]_L &= \alpha^*_{\Gamma,\Delta}; ([\![\nabla : (\Gamma \vdash M{:}\,(\theta' \multimap \theta))]\!]_L \otimes \\
&\qquad\qquad [\![\nabla' : (\Delta \vdash N{:}\,\theta')]\!]_L); \alpha^{*-1}_{\Gamma,\Delta} \\
[\![Tens(\nabla,\nabla') : (\Gamma :: \Delta \vdash M \otimes N{:}\,(\theta \otimes \theta'))]\!]_L &= \alpha^*_{\Gamma,\Delta}; ([\![\nabla : \Gamma \vdash M{:}\,\theta]\!]_L \otimes \\
&\qquad \alpha^*_{\Gamma,\Delta}; ([\![\nabla' : \Delta \vdash N{:}\,\theta']\!]_L); \alpha^{*-1}_{\Gamma,\Delta} \\
[\![Weak_{x:\theta'}(\nabla) : (\Gamma :: x{:}\,\theta' \vdash M{:}\,\theta)]\!]_L &= (\mathbf{id}_{[\![\Gamma]\!]} \otimes \top_{[\![\theta']\!]}); \rho_{[\![\Gamma]\!]}; [\![\nabla : (\Gamma \vdash M{:}\,\theta)]\!]_L \\
[\![Exch_{y,x}(\nabla) : (\Gamma :: x{:}\,\theta' :: y{:}\,\theta'' :: \Delta \vdash M{:}\,\theta)]\!]_L &= \gamma^*_{\Gamma,\theta',\theta'',\Delta}; [\![\nabla : (\Gamma :: y{:}\,\theta'' :: x{:}\,\theta' :: \Delta \vdash M{:}\,\theta)]\!]_L \\
[\![Abs_x(\nabla) : (\Gamma :: \Delta \vdash \lambda(x{:}\,\theta').M{:}\,(\theta' \multimap \theta))]\!]_L &= \mathbf{id}_{[\![\Gamma :: \Delta]\!]}
\end{aligned}
$$

$$
\begin{aligned}
[\![\nabla : (\Gamma \vdash M{:}\,\theta)]\!]_R &: [\![\mathbf{fv}(\nabla)]\!] \longrightarrow [\![\theta]\!] \\
[\![Id_{x:\theta} : (x{:}\,\theta \vdash x{:}\,\theta)]\!]_R &= \mathbf{id}_{[\![\theta]\!]} \\
[\![Const_M : (\vdash M{:}\,\theta)]\!]_R &= [\![\vdash M : \theta]\!] \\
[\![App(\nabla,\nabla') : (\Gamma :: \Delta \vdash MN{:}\,\theta)]\!]_R &= \alpha^*_{\Gamma,\Delta}; ([\![\nabla : (\Gamma \vdash M{:}\,(\theta' \multimap \theta))]\!]_R \otimes \\
&\qquad\qquad [\![\nabla' : (\Delta \vdash N{:}\,\theta')]\!]_R); \mathbf{eval}_{[\![\theta']\!],[\![\theta]\!]} \\
[\![Tens(\nabla,\nabla') : (\Gamma :: \Delta \vdash M \otimes N{:}\,(\theta \otimes \theta'))]\!]_R &= \alpha^*_{\Gamma,\Delta}; [\![\nabla : (\Gamma \vdash M{:}\,\theta)]\!]_R \otimes [\![\nabla' : (\Delta \vdash N{:}\,\theta')]\!]_R \\
[\![Weak_x(\nabla) : (\Gamma :: x{:}\,\theta' \vdash M{:}\,\theta)]\!]_R &= [\![\nabla : (\Gamma \vdash M{:}\,\theta)]\!]_R \\
[\![Exch_{y,x}(\nabla) : (\Gamma :: x{:}\,\theta' :: y{:}\,\theta'' :: \Delta \vdash M{:}\,\theta)]\!]_R &= [\![\nabla : (\Gamma :: y{:}\,\theta'' :: x{:}\,\theta' :: \Delta \vdash M{:}\,\theta)]\!]_R \\
[\![Abs_x(\nabla) : (\Gamma :: \Delta \vdash \lambda(x{:}\,\theta').M{:}\,(\theta' \multimap \theta))]\!]_R &= \mathbf{abs}^*_{[\![\Gamma]\!],[\![\theta']\!],[\![\Delta]\!],[\![\theta]\!]}([\![\nabla : (\Gamma :: x{:}\,\theta' :: \Delta \vdash M{:}\,\theta)]\!])
\end{aligned}
$$

Figure 5.2.5: Definitions used by Lemma 5.2.1

∇ is not Abstraction, then the condition can be fulfilled by replacing each subtree ∇' of ∇ with ∇'_c. Thus, the only interesting case is where the last rule applied in ∇ is an Abstraction:

$$\frac{\Gamma_1 :: x\!:\theta_2 :: \Gamma_2 \vdash M_1 : \theta_1}{\Gamma_1 :: \Gamma_2 \vdash \lambda(x\!:\theta_2).M_1 : (\theta_2 \multimap \theta_1)}$$

Thus, without loss of generality (due to the inductive hypothesis), we can assume that apart from possibly premises of the root of ∇, all premises of Abstraction rules in ∇ are the root of subtrees $\nabla'' : (\Gamma'' \vdash M'' : \theta'')$ for which $\Gamma'' = \mathbf{fv}(\nabla'')$. There clearly must be some derivation of $\Gamma_1 :: x : \theta_2 :: \Gamma_2 \vdash M_1 : \theta_1$ that contains $\mathbf{fv}(\Gamma_1 :: x : \theta_2 :: \Gamma_2 \vdash M_1 : \theta_1) \vdash M_1 : \theta_1$, because $\mathbf{fv}(\Gamma_1 :: x : \theta_2 :: \Gamma_2 \vdash M_1 : \theta_1)$ can easily be seen via induction to contain no elements that are not in $\Gamma_1 :: x : \theta_2 :: \Gamma_2$, and thus it is possible to apply Weakening and Exchange to it in order to make the context the same while leaving the term and its type alone. Thus, without loss of generality (due to Lemma 5.2.1), we can assume that $\mathbf{fv}(\Gamma_1 :: x : \theta_2 :: \Gamma_2 \vdash M_1 : \theta_1) \vdash M_1 : \theta_1$ appears somewhere in ∇ (and the only rules that appear below it in ∇ must be Weakening or Exchange, because all other rules would increase the number of symbols in the term, and there is no rule that would allow decreasing the number of symbols, thus it would be impossible for M_1 to appear as the term ever again, but we assumed that M_1 appears at the base of ∇).

It is thus sufficient to prove that Weakening and Exchange both commute with Abstraction. For Weakening, we need these trees to have equal denotations:

$$\frac{\dfrac{\Gamma_1 :: x\!:\theta_2 :: \Gamma_2 \vdash M_1 : \theta_1}{\Gamma_1 :: \Gamma_2 \vdash \lambda(x\!:\theta_2).M_1 : (\theta_2 \multimap \theta_1)}}{\Gamma_1 :: \Gamma_2 :: y : \theta_3 \vdash \lambda(x\!:\theta_2).M_1 : (\theta_2 \multimap \theta_1)} \qquad \frac{\dfrac{\Gamma_1 :: x\!:\theta_2 :: \Gamma_2 \vdash M_1 : \theta_1}{\Gamma_1 :: x : \theta_2 :: \Gamma_2 :: y : \theta_3 \vdash M_1 : \theta_1}}{\Gamma_1 :: \Gamma_2 :: y : \theta_3 \vdash \lambda(x\!:\theta_2).M_1 : (\theta_2 \multimap \theta_1)}$$

Let ∇ be the tree that derives the premise of these trees. Expanding the definitions, we discover that we need to prove

$$\begin{aligned}&(\mathbf{id}_{[\![\Gamma::\Delta]\!]} \otimes \top_{[\![\theta_3]\!]}); \rho_{[\![\Gamma::\Delta]\!]}; \mathbf{abs}^*_{[\![\Gamma_1]\!],[\![\theta_2]\!],[\![\Gamma_2]\!],[\![\theta_1]\!]}([\![\nabla]\!]) \\ &= \mathbf{abs}^*_{[\![\Gamma_1]\!],[\![\theta_2]\!],[\![\Gamma_2::y:\theta_3]\!],[\![\theta_1]\!]}((\mathbf{id}_{[\![\Gamma::x:\theta_2::\Delta]\!]} \otimes \top_{[\![\theta_3]\!]}); \rho_{[\![\Gamma::x:\theta_2::\Delta]\!]}; [\![\nabla]\!])\end{aligned}$$

which is true due to naturality.

For Exchange, we do not need to expand the definitions, because we can once again use existing results. Kelly and MacLane proved in [27, Theorem 2.4 on page 107] that two natural transformations, formed solely out of α, α^{-1}, γ, ρ, ρ^{-1}, **abs**, **eval**, **id**, $\Rightarrow$, $\otimes$, and morphism composition, must be equal if they are between the same functors and parameterized on the right hand side of every $\Rightarrow$, conditions that are satisfied by our definitions of Exchange and Abstraction. (Weakening does not satisfy this definition due to the use of $\top$, which is why a separate proof was needed for that case.)

Thus, in all cases, the statement of the lemma holds. $\square$

Theorem 5.2.3. *Given two derivation trees* $\nabla : (\Gamma \vdash M : \theta)$, $\nabla' : (\Gamma \vdash M : \theta)$, *then* $[\![\nabla]\!] = [\![\nabla']\!]$.

Proof. By Lemma 5.2.2, there exist derivation trees ∇_c, ∇'_c that derive $\Gamma \vdash M : \theta$ such that $[\![\nabla]\!] = [\![\nabla_c]\!]$, $[\![\nabla']\!] = [\![\nabla'_c]\!]$, and such that ∇_c, ∇'_c each use Abstraction only where the subtree $\nabla'' : (\Gamma'' \vdash M'' : \theta'')$ rooted at the premise of that Abstraction has $\Gamma'' = \mathbf{fv}(\nabla'')$. Lemma 5.2.1 proves that $[\![\nabla_c]\!] = [\![\nabla'_c]\!]$; thus $[\![\nabla]\!] = [\![\nabla']\!]$. $\square$

It is also possible to produce a concrete semantics for Affine ICA via various methods. For example, I have worked on producing an instance of this category via game semantics; although this work is complete, it is not included here, for space reasons and because it would distract from the main flow of this thesis. One advantage of producing the concrete semantics is that it showed directly that Affine ICA is finite-state (via giving, for each term, an explicit finite set of states that that term could be in). However, due to the lack of contraction, this is relatively obvious even without the semantics: the lack of contraction and recursion means that each term can only execute once, and thus the length of time for which any given program executes is bounded by the number of terms it contains. A program cannot use infinite amounts of state in a finite time.

Chapter 6

REUSING CODE VIA COPYING

One of the major problems behind Affine ICA is that it has no contraction at all, meaning that there is no way to reuse code; each term can be evaluated in only one context. (The other, larger problem is that each term can only be evaluated once, meaning that all programs trivially terminate; we consider that problem in Chapter 8.) In this chapter, we first consider the topic of *bounded contraction*: contraction in which there is an explicit, finite bound on the number of times the term being contracted will execute. We also consider a possible technique for implementing bounded contraction in a finite-state implementation of a language: via *copying* the terms being contracted.

6.1 Bounded contraction

The main reason that the contraction rule of full ICA causes problems when a finite-state language is required is that it does not place any limitation on how much a term can be contracted: it is easy to design a term (e.g. $\mathsf{fix}(\lambda f.\mathsf{par}(x)(f(x)))$) which runs infinitely many copies of a term (x in this example) in parallel, and hard to see how such a term could possibly be considered finite-state. A good example is that of hardware synthesis, the direction that originally inspired this work: there are practical limits to the number of circuits that can be fit onto a physical silicon chip, meaning that this sort of infinite contraction cannot be implemented in hardware. There have nonetheless been attempts to implement something similar to this, such as the implementation by Ferizis in [13] that uses reconfigurable hardware so that the circuits can be created at runtime when they become necessary, rather than needing to all be synthesized in advance; however, a more generally applicable method is to design a type system to prevent the problem arising in the first place.

It might seem as though the issues with unbounded contractions in ICA stem from the fix constant, but Ghica, Murawski and Ong showed in [18, Theorem 6 on page 8] that even after removing fix from ICA, program equivalence is nonetheless undecidable; and if program equivalence is undecidable, it immediately follows that not all programs can be implemented as state machines (for which equivalence is decidable, e.g. as shown in [26] by Hopcroft and Karp). (The proof in [18] cannot directly be used for this purpose in that it requires a non-halting ("diverging") command Ω_{com}, which it defines as $\mathsf{fix}(\lambda x : \mathsf{com}.x)$, and thus as written requires a fix constant in the language. However, it can be corrected to work in ICA-minus-fix via using a different definition for Ω_{com} that does not involve recursion. For example, $\mathsf{newsem}(\lambda s.\mathsf{seq}(\mathsf{grab}(s))(\mathsf{grab}(s)))$ always diverges due to the deadlock on the semaphore s.) They identify the problem as not being due to recursion or loops in their own right, but rather the fact that there is no limit on how many times a function might use its argument. Even if it is known that a function g contains no recursion or loops, that still gives no information about how many times x might be evaluated in $g(x)$ (except that it is bounded by some unknown finite integer); and for undecidability proofs, an unbounded number of evaluations works just as well as an infinite number of evaluations, because if a term $g(x)$ halts at all, it must halt after some finite number of evaluations of x, and with no requirements on how often x is used, it always has the potential to be greater than the number of evaluations required for the term to halt.

The usual solution to this problem, then, is to require that some explicit number be given, in the type of a function, for the number of times it can use its argument. For example, after proving ICA-minus-fix undecidable, the authors of the above proof gave a type system Syntactic Control of Concurrency (SCC) that uses types that look like $(\mathsf{com}^4 \to \mathsf{com})^3 \to \mathsf{com}$ (meaning "a function returning com whose argument (a function taking and returning com that may use its argument up to four times) is used at most three times"). An earlier type system, Bounded Linear Logic (BLL) [23] by Girard et al, provided a similar solution to this problem, with a different notation ("$!_3(!_4\mathsf{com} \to \mathsf{com}) \to \mathsf{com}$" is suggested in the paper for this type, but BLL normally uses more complex types that imply substitutions in the terms). Both type systems have extra complexities of their own, though. SCC adds SCI-like handling of sequential composition, something that this thesis considers a separate problem from that of bounded contraction, and which will be discussed in Chapter 8. Meanwhile, BLL has much more general types, allowing types along the lines of $\mathsf{com} \to !_2\mathsf{com}$ that do not correspond to anything in SCC, in order to be able to calculate bounds for calling conventions other than just call-by-name. Dealing with this extra power in the type systems would distract from the main arguments in this section, so instead, we will present the results of this chapter using a version of Affine ICA with bounded contraction, called Bounded ICA.

Bounded ICA, shown in Figure 6.1.1, is formed from SCC via the use of an Affine-ICA-like Tensor rule (and uncurry constant), and via fixing a mistake in the definition of Subtyping in the original paper (a mistake that was corrected in future papers, such as [20] by Ghica and myself, and thus can be considered to not be part of the definition of SCC). (The existing sources on SCC vary as to whether it contains products, and on the details if it does; Bounded ICA does not, and has tensors as its only pair-like structure.) Apart from that, the two type systems are identical down to notation. (The notation used here was chosen for consistency with other type systems that will be introduced later in this thesis.) The constants are the same as those of Affine ICA, with two exceptions: constants that are not of base type have bounds added, and newvar's type is based on its type from ICA, rather than its type from Affine ICA, because the use of bounded contraction means that we can contract the arguments used for reading and writing, rather than needing a separate callback for each read or write that is made to the variable. We use the type $\mathsf{exp}_J \otimes (1 \cdot \mathsf{exp}_J \multimap \mathsf{com})$ as an effective definition for var, because it simplifies things considerably to have all the base types work the same way, and it also means that deref and assign have simple implementations in terms of projections (which in turn can be implemented in terms of uncurry), and thus do not need to be separate constants. The constants are shown in Figure 6.1.2.

The main changes from Affine ICA are the addition of the bounds in the function types,

$$\theta ::= \mathsf{exp}_J \mid j \cdot \theta \multimap \theta \mid \theta \otimes \theta, \text{ where } J \subset \mathbb{N} \text{ is a finite set, } j \in \mathbb{N} \text{ is a nonnegative integer}$$

$$\Gamma ::= \text{sequence of } x\!:j \cdot \theta, \text{ where } j \in \mathbb{N} \text{ is a nonnegative integer}$$

$$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$

$$j \cdot \varepsilon \triangleq j;\ j \cdot (x\!:j' \cdot \theta :: \Gamma) \triangleq jj' \cdot \theta :: j \cdot \Gamma$$

$$\frac{x \neq y}{x\!:\theta \# y\!:\theta'} \qquad \frac{\Gamma \# \Delta \qquad \Gamma' \# \Delta}{\Gamma :: \Gamma' \# \Delta} \qquad \frac{x\!:\theta \# \Gamma}{x \# \Gamma}$$

$$\frac{}{\mathsf{exp}_J \leq \mathsf{exp}_J} \qquad \frac{j_1 \leq j_2 \qquad \theta_2' \leq \theta_1' \qquad \theta_1 \leq \theta_2}{j_1 \cdot \theta_1' \multimap \theta_1 \leq j_2 \cdot \theta_2' \multimap \theta_2} \qquad \frac{\theta_1 \leq \theta_2 \qquad \theta_1' \leq \theta_2'}{\theta_1 \otimes \theta_1' \leq \theta_2 \otimes \theta_2'}$$

$$\frac{}{x\!:1 \cdot \theta \vdash x\!:\theta}\ \text{Identity}$$

$$\frac{\Gamma \vdash M\!:\theta \qquad \theta \leq \theta'}{\Gamma \vdash M\!:\theta'}\ \text{Subtyping}$$

$$\frac{M\!:\theta \text{ is a constant}}{\vdash M\!:\theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M\!:\theta \qquad x \# \Gamma}{\Gamma :: x\!:j \cdot \theta' \vdash M\!:\theta}\ \text{Weakening}$$

$$\frac{\Gamma :: x\!:j' \cdot \theta' :: y\!:j'' \cdot \theta'' :: \Delta \vdash M\!:\theta}{\Gamma :: y\!:j'' \cdot \theta'' :: x\!:j' \cdot \theta' :: \Delta \vdash M\!:\theta}\ \text{Exchange}$$

$$\frac{\Gamma :: x\!:j \cdot \theta' :: \Delta \vdash M\!:\theta}{\Gamma :: \Delta \vdash \lambda(x\!:j \cdot \theta').M\!:(j \cdot \theta' \multimap \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M\!:(j \cdot \theta' \multimap \theta) \qquad \Delta \vdash N\!:\theta' \qquad \Gamma \# \Delta}{\Gamma :: j \cdot \Delta \vdash MN\!:\theta}\ \text{Application}$$

$$\frac{\Gamma \vdash M\!:\theta \qquad \Delta \vdash N\!:\theta' \qquad \Gamma \# \Delta}{\Gamma :: \Delta \vdash M \otimes N\!:(\theta \otimes \theta')}\ \text{Tensor}$$

$$\frac{\Gamma :: x\!:j \cdot \theta' :: y\!:j' \cdot \theta' \vdash M\!:\theta}{\Gamma :: x\!:(j + j') \cdot \theta' \vdash M[x/y]\!:\theta}\ \text{Contraction}$$

Figure 6.1.1: Bounded ICA

$$\begin{array}{rcl}
\mathsf{skip} & : & \mathsf{com} \\
j_J & : & \mathsf{exp}_J \\
\mathsf{seq}_J & : & 1\cdot\mathsf{com} \multimap 1\cdot\mathsf{exp}_J \multimap \mathsf{exp}_J \\
\mathsf{par} & : & 1\cdot\mathsf{com} \multimap 1\cdot\mathsf{com} \multimap \mathsf{com} \\
\mathsf{if} & : & 1\cdot\mathsf{exp}_{\{0,1\}} \multimap 1\cdot\mathsf{exp}_J \multimap 1\cdot\mathsf{exp}_J \multimap \mathsf{exp}_J \\
\mathsf{uncurry}_{j,\theta'_1,\theta'_2,\theta} & : & 1\cdot(j\cdot\theta'_1 \multimap (j\cdot\theta'_2 \multimap \theta)) \multimap (j\cdot(\theta'_1\otimes\theta'_2) \multimap \theta) \\
\mathsf{op}_{J,\bullet} & : & 1\cdot\mathsf{exp}_J \multimap 1\cdot\mathsf{exp}_J \multimap \mathsf{exp}_J \\
\mathsf{newvar}_{j,J} & : & 1\cdot(j\cdot(\mathsf{exp}_J\otimes(1\cdot\mathsf{exp}_J \multimap \mathsf{com})) \multimap \mathsf{com}) \multimap \mathsf{com}
\end{array}$$

Figure 6.1.2: Constants of Bounded ICA

and the addition of two new rules, Subtyping and Contraction. The Contraction rule can be understood as meaning that if an open term takes two free variables of the same type θ, one of which is used j times and the other of which is used j' times, it is possible to use the same free variable to satisfy both of these requirements, but it will then be used $j+j'$ times. The Subtyping rule is taken from SCC, where its purpose (according to SCC's authors in [18, page 10]) is to implement the intuition that if a function needs fewer copies of an argument than are available, or is able to give its argument more copies of an argument than are needed, this should not cause the term to fail to type. Bounded ICA features the same rule, so that results about Bounded ICA can also be applied to SCC; in particular, the the Subtyping rule and its shortcomings are discussed in Sections 6.3 (which finds that it is insufficient to type every term that would be expected to type in SCC) and 6.5 (which shows that in the presence of some reasonable equivalences between terms, it is unnecessary).

6.2 Type inference in Bounded ICA

The motivation of SCC was to produce decidable program equivalence, something at which it succeeds; equivalence of Bounded ICA programs is decidable for the same reasons. There would only be minimal (tensor-related) benefit in reproducing the existing proofs from [18] here. Instead, we focus on a different problem. The purpose of SCC is to ensure that all contractions in a term are bounded; and it is not always obvious whether for any given ICA term, all contractions are bounded. This immediately raises two questions: can we find an algorithm for determining whether an ICA term has an SCC type (or a Bounded ICA type)? What about determining whether an ICA term uses only bounded contraction, and is that equivalent to the previous question? In this section, we will focus on the first question; the second question will be discussed later in this thesis, in Section 6.3.

If it is desired to use SCC, or Bounded ICA, as an intermediary in a compilation process (i.e. implementing a program via finding an SCC type for that program, then using a semantics of SCC as part of the implementation), then a type inference algorithm for SCC is particularly important; with a type system as complex as that of SCC or Bounded ICA, the standard type inference techniques used for lambda calculus are insufficient, and it would be unreasonable to require a programmer to specify all the bounds manually. Instead, for it to be reasonable to use SCC or Bounded ICA as a source language that people program in, we would want an algorithm that automatically calculates the bounds. The most obvious method is via a type inference that uses unknowns as the bounds, and produces constraints between them; for example, the type of $f, x \vdash f(x)$ could be inferred as $f : j_1 \cdot (j_2 \cdot \theta' \multimap \theta), x : j_3 \cdot \theta' \vdash f(x) : \theta$ with a constraint that $j_2 \leq j_3$.

All the results in this section apply to either Bounded ICA, or to SCC without constants; the difference between Bounded ICA's Tensor rule and SCC's Product rule is not particularly relevant in any of the proofs, because ICA can be expressed in a form that uses explicit contraction for function applications but implicit contraction for products. Allowing for SCC's constants presents problems unrelated to the results of this section, though, because they require SCC programs to be written syntactically differently from ICA programs; $\mathsf{seq}(\mathsf{skip})(\mathsf{skip})$ is a well-typed ICA program, but the equivalent program in SCC has to be written $\mathsf{seq}(\langle \mathsf{skip}, \mathsf{skip} \rangle)$. (The only problem with SCC's constants is that they have structurally different types from ICA's; the results in this section apply to SCC if "ICA" is consistently replaced with "ICA with SCC's constants".) We discuss this shortcoming much later in this thesis, in Section 8.1; for the time being, we thus focus on Bounded ICA, to keep the presentation simple, with the understanding that similar results exist for SCC.[1]

Before starting, we need to define precisely what we mean by type inference. The syntax of a term contains the type of each lambda abstraction, e.g. $\lambda(x : \mathsf{com}).x$ and $\lambda(x : \mathsf{exp}_{\{0,1\}}).x$ are different terms in ICA. However, this makes it impossible to find a "Bounded ICA type of an ICA term", etc., because the types have different syntax and so the terms do too. Thus, in the rest of this section, we will sometimes write a judgement using an expanded syntax $\Gamma \vdash M, \Theta : \theta$; here, M is considered to be a term with holes for the types of lambdas (e.g. $\lambda(x : -).x$), and Θ specifies which types go in each of those holes. We call this new M an "uninferred term", and say that an uninferred term M types in a type system if some Γ, Θ, θ exist such that $\Gamma \vdash M, \Theta : \theta$ types in that type system. The $\Gamma \vdash M, \Theta : \theta$ and $\Gamma \vdash M : \theta$ syntaxes are equivalent and interchangeable; thus, when drawing derivation trees, we will typically write a judgement as, e.g., $x : \mathsf{com} \vdash \lambda(y : \mathsf{com}).\mathsf{seq}(x)(y) : \mathsf{com} \multimap \mathsf{com}$ rather than $x : \mathsf{com} \vdash \lambda(y : -_1).\mathsf{seq}(x)(y), \{-_1 \mapsto \mathsf{com}\} : \mathsf{com} \multimap \mathsf{com}$ (the notation would need to place subscripts onto the holes so that they can be distinguished from each other), because it is shorter and both syntaxes convey the same information.

The first step in Bounded ICA type inference, then, is to perform ICA type inference on the term; we produce a type for the term in ICA with explicit contractions (the type system shown earlier in this thesis in Figure 5.1.1). A comparison of Figure 5.1.1 and Figure 6.1.1 shows that the only differences between them are the difference in notation between Products and Tensors (the rules are identical, just one uses the symbol $\times$ and the other uses the symbol $\otimes$), a difference which we do not concern ourselves with because differences in notation are trivial to work around, and the presence of bounds in the Bounded ICA term. The important fact about the bounds is that they do not expand which terms are legal:

[1]I first published the general result of this section in [20], as joint work with my supervisor Dan Ghica, with respect to SCC rather than Bounded ICA. The result has been converted to a Bounded ICA setting for this thesis, and greatly expanded (with more formality in the proofs, and a proof of completeness as well as soundness). Although the paper was joint work, and contained results from both authors, this section happens to only include results that were originally mine, even in the SCC setting.

Theorem 6.2.1. *Given any Bounded ICA derivation, deleting all bounds from it (to produce terms that follow the syntax of ICA, rather than of Bounded ICA), and all uses of the Subtyping rule, produces an ICA derivation with explicit contraction.*

Proof. All the rules of Bounded ICA produce a rule of ICA when bounds are deleted, apart from the Subtyping rule, which becomes trivial (deriving a judgement from itself) when bounds are deleted. Thus, the result is trivial by structural induction. □

Corollary 6.2.2. *If an uninferred term is not derivable in ICA, it is not derivable in Bounded ICA either.*

Thus, if a term has a Bounded ICA derivation, this derivation must be producible via starting with an ICA derivation from the term, then adding in bounds and uses of the Subtyping rule. As a simplification, we note that deriving a term from itself using Subtyping is always legal (by structural induction), and that two applications of Subtyping in a row can always be replaced with one combined application (because $\leq$ on types is transitive). Thus, we can merge the Subtyping rule into each of the other rules of Bounded ICA, and have an equivalent type system. The idea, then, is to infer constraints on the bounds for each ICA derivation of a term, and see whether any of those sets of constraints are solvable.

If we wish for our type inference algorithm to terminate, however, we immediately hit a problem: a term can have infinitely many ICA derivations. Luckily, though, this problem is not too hard to work around. Obviously, whether or not a term types in Bounded ICA, there is some number j such that if there is no derivation of the term using no more than j productions, there is no derivation of the term at all (an arbitrary j works for this if the term does not type, the number of productions in an arbitrary derivation of the term works if it does type). What we need to prove is not merely that j exists (which is obvious), but that it is computable (i.e. an algorithm exists to calculate a specific value of j):

Theorem 6.2.3. *Given any uninferred term M, it is possible to compute a number j such that if there is a Bounded ICA derivation of M, at least one such derivation uses no more than j productions.*

Proof. We show that $j = (k+1)(l+m)(((3l+k)!+1)(4l+k+1)+1)$ is such a number, where k is the total number of uses of λ, function application, and tensor formations that appear in M, l is the number of times a variable is mentioned in M, and m is the number of constants that appear in M. Let ∇ be a Bounded ICA derivation of M with the fewest number of productions (if ∇ does not exist, the theorem is trivial). We prove by contradiction that ∇ has no more than j productions, via showing that if ∇ had more than j productions, we could find a series of subtrees of ∇ with certain properties, the last of which obviously cannot be contained in ∇.

First, we note that the number of Identity productions in ∇ cannot exceed l, the number of Constant productions in ∇ cannot exceed m, and the total number of Abstraction, Application, and Tensor productions in ∇ cannot exceed k (because for each production, the terms in each of its premises appear in its result, unchanged apart from variable names). Define a *chain* as a sequence of productions, the first of which is an axiom, and for which all but the first have as a premise the consequence of the production before. Because each chain starts with an axiom, the number of chains cannot exceed $l+m$. Each production belongs to at least one chain; thus at least one chain must have at least $(k+1)(((3l+k)!+1)(4l+k+1)+1)+1$ elements (because there are more than j productions in total). Consider the last $((3l+k)!+1)(4l+k+1)+1$ elements of the chain, the $((3l+k)!+1)(4l+k+1)+1$ elements that precede those, and so on; at least $k+1$ such blocks of elements exist in the chain. Such blocks cannot contain axioms (because the only axiom in the chain is the first element), and at least one block contains no uses of Abstraction, Application, or Tensor productions (because there are at most k such productions but $k+1$ blocks). Thus, this block is a sequence of $((3l+k)!+1)(4l+k+1)+1$ elements, each of which is (by elimination) Subtyping, Weakening, Exchange, or Contraction.

We next note that Weakening, Exchange, and Contraction do not change the type of the term within the judgement (the θ part of $\Gamma \vdash M' : \theta$), and are entirely parametric on it; meanwhile, Subtyping changes only the type of the term, and is entirely parametric on the rest of the judgement. Thus, we can assume without loss of generality that all uses of Subtyping appear at the end of the block. Also, two uses of Subtyping cannot appear in a row, because then they could be combined into a single use of Subtyping ($\leq$ on types is transitive), contradicting the assumption that ∇ has the fewest number of productions among derivations of M. Thus, there is a sequence of $((3l+k)!+1)(4l+k+1)$ consecutive elements within the block, each of which is Weakening, Exchange, or Contraction.

For each use of Contraction anywhere in ∇, either it reduces the number of distinct variable names that appear within the term, or it doesn't. The number of Contractions that reduce the number of distinct variable names cannot exceed l (because there are only l uses of variable names in M, and after reducing the number of distinct variable names, all the uses of either variable will end up in the final term). For other Contractions, we note that it is impossible for both variables being contracted to have been created via Weakening (otherwise, both the contraction and one of the weakenings, as well as any exchanges on the variable whose weakening is deleted, can be deleted from ∇ without changing anything apart from the bound on the other weakening, leaving a derivation of M with fewer productions). It is also impossible for any variable to be contracted twice with neither contraction reducing the number of distinct variable names that appear in the term (again, one of the weakenings and one of the contractions can be deleted via increasing the bound on the other weakening). Thus, the number of Contractions that appear in ∇ cannot exceed $2l$.

For each use of Weakening anywhere in ∇, either the variable being weakened is removed from the context via Abstraction, or it is removed from the context via Contraction, or it remains in the context at the end of the derivation. The last case is impossible, because then the weakening (and any exchanges on the weakened variable) could be removed and still lead to a derivation of M (with a different context, but that does not matter to the statement of the theorem). The other two cases allow for at most k and $2l$ uses of Weakening respectively. No context anywhere in the derivation can thus have more than $3l+k$ variables in it (the maximum total number of Identity and Weakening uses in ∇).

We next consider our block of $((3l+k)!+1)(4l+k+1)$ elements, each of which is Weakening, Exchange or Contraction. Just as the chain was divided into blocks, we can divide the block into sub-blocks of $(3l+k)!+1$ elements each; because there are $4l+k+1$ such sub-blocks, and at most $(2l+k)+2l=4l+k$ uses of Weakening or Contraction, one sub-block must be made entirely out of Exchange productions. But there are at most $(3l+k)!$ possible permutations of the context; thus, two of the judgements within that sub-block must be the same, allowing all the production rules between those judgements to be deleted to produce a derivation of M with fewer production rules than ∇. This is a contradiction, so ∇ must have no more than j productions. $\square$

Even bounding the number of productions in a derivation, we still have the potential for infinitely many ICA derivations of a term; although there are only finitely many shapes for the derivation tree, terms like $\lambda x.x$ have infinitely many possible types (for example, $\mathsf{com} \to \mathsf{com}$ or $(\mathsf{exp}_{\{0,1\}} \to \mathsf{exp}_{\{0,1\}}) \to (\mathsf{exp}_{\{0,1\}} \to \mathsf{exp}_{\{0,1\}})$). We need to show that this also does not make a difference:

Theorem 6.2.4. *Given any uninferred term M, it is possible to compute a number k such that if there are any Bounded ICA derivations of M, some such derivation ∇ which has no more productions than any other such derivation contains no type that includes more than k occurrences of base types.*

Proof. By Theorem 6.2.3, we can compute a number j such that ∇ has no more than j productions. All the rules of Bounded ICA are entirely parametric on the types of the term and on free variables in the context, apart from Abstraction and Application (which each require one type to be of the form $l \cdot \theta' \multimap \theta$), Tensor (which requires one type to be of the form $\theta \otimes \theta'$), Subtyping (which only changes bounds), and Constant (which cannot produce a type with any more than 6 base types). Thus, if any type θ anywhere in ∇ contains more than $6j+1$ base types, we can identify some non-base type θ' within θ such that all the productions in ∇ are parametric on the value of θ', and consistently replace θ' with a base type of our choice (such as com). This process can be carried out repeatedly until ∇ has no type containing more than $6j+1$ base types, so we take $k=6j+1$, fulfilling the statement of the theorem. $\square$

This forms the first half of our algorithm for determining whether a term has a Bounded ICA type; we find all derivations for it with no more than j productions and no more than k occurrences of base types in any type in the derivation, which is a finite number of derivations if derivations that differ only in variable names are considered equal. In fact, this means that we do not technically need an ICA inference algorithm to prove that Bounded ICA typing is decidable, because we could just enumerate every possible derivation tree within these bounds and check them for well-formedness. Decidability results care about the existence of an algorithm, not how efficient it is. (That said, there are much more efficient ways to do Bounded ICA type inference in practice; it's merely harder to prove that they are complete, i.e. they always find a type if such a type exists.)

Next, we give an intermediate type system that serves as a step in the inference between ICA and Bounded ICA. (This sort of algorithm is more commonly given as a set of rules for deriving constraints from a tree, but such sets of rules are quite close to a type system anyway, and so it seems more sensible to formalize it as a type system than using some more ad hoc method.) This intermediate type system is shown in Figure 6.2.1. The idea is to use a four-part judgement $\Gamma \vdash M : \theta \triangleright K$, where the bounds on types are not integers j as in Bounded ICA, but rather values τ drawn from an arbitrary countably infinite set $\mathbb{A}$; each τ represents a specific unknown bound (with bounds represented by the same τ having the same value; bounds represented by different τ might or might not have the same value). K, the newly introduced part of the judgement, is a set of constraints κ of the forms $\tau \geq \tau$, $\tau \geq \tau\tau$, $\tau \geq \tau + \tau$, and $\tau \geq j$ (chosen because those forms suffice for Bounded ICA and SCC inference). The basic idea behind this construction is that this intermediate type system is equivalent to ICA (in that it types the same terms), and that by adding a side condition (that the set K of constraints is solvable), it becomes equivalent to Bounded ICA instead.

We do not need to require that the various τ are fresh. All we typically care about is that some derivation exists; thus, derivations that conflate two τ that need not be conflated are harmless, because the derivation that respects reasonable freshness rules for τ also exists. Derivations

$\theta ::= \mathsf{exp}_J \mid \tau\cdot\theta \multimap \theta \mid \theta\otimes\theta$, where $J\subset\mathbb{N}$ is a finite set, $\tau\in\mathbb{A}$, $\mathbb{A}$ is any countably infinite set

$\Gamma ::=$ sequence of $x\!:\!j\cdot\theta$, where $j\in\mathbb{N}$ is a nonnegative integer

$\kappa ::= \tau\geq j \mid \tau\geq\tau \mid \tau\geq\tau+\tau \mid \tau\geq\tau\tau$, for $j\in\mathbb{N}$, $\tau\in\mathbb{A}$

$K ::=$ set of κ

$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$

$$\frac{x\neq y}{x\!:\!\theta\#y\!:\!\theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma::\Gamma'\#\Delta} \qquad \frac{x\!:\!\theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{\mathsf{exp}_J\leq\mathsf{exp}_J\rhd\emptyset} \qquad \frac{\theta_1\leq\theta_2\rhd K \qquad \theta_1'\leq\theta_2'\rhd K'}{\theta_1\otimes\theta_1'\leq\theta_2\otimes\theta_2'\rhd K\cup K'}$$

$$\frac{\theta_2'\leq\theta_1'\rhd K' \qquad \theta_1\leq\theta_2\rhd K}{\tau_1\cdot\theta_1'\multimap\theta_1\leq\tau_2\cdot\theta_2'\multimap\theta_2\rhd K'\cup K\cup\{\tau_2\geq\tau_1\}}$$

$$\frac{}{\tau\cdot(\tau_1\cdot\theta)\leq\tau_2\cdot\theta\rhd\{\tau_2\geq\tau\tau_1\}} \qquad \frac{\tau\cdot\Gamma_1\leq\Gamma_2\rhd K \qquad \tau\cdot\Gamma_1'\leq\Gamma_2'\rhd K'}{\tau\cdot(\Gamma_1::\Gamma_1')\leq\Gamma_2::\Gamma_2'\rhd K\cup K'}$$

$$\frac{\theta_L\leq\theta_R\rhd K}{x\!:\!\tau\cdot\theta_L\vdash x\!:\!\theta_R\rhd K\cup\{\tau\geq 1\}}\ \text{Identity}$$

$$\frac{M\!:\!\theta_I\rhd K_1 \text{ is a constant} \qquad \theta_I\leq\theta_O\rhd K_2}{\vdash M\!:\!\theta_O\rhd K_1\cup K_2}\ \text{Constant}$$

$$\frac{\Gamma\vdash M\!:\!\theta_I\rhd K_1 \qquad x\#\Gamma \qquad \theta_I\leq\theta_O\rhd K_2}{\Gamma::x\!:\!\tau\cdot\theta_W\vdash M\!:\!\theta_O\rhd K_1\cup K_2}\ \text{Weakening}$$

$$\frac{\Gamma::x\!:\!\tau'\cdot\theta'::y\!:\!\tau''\cdot\theta''::\Delta\vdash M\!:\!\theta_I\rhd K_1 \qquad \theta_I\leq\theta_O\rhd K_2}{\Gamma::y\!:\!\tau''\cdot\theta''::x\!:\!\tau'\cdot\theta'::\Delta\vdash M\!:\!\theta_O\rhd K_1\cup K_2}\ \text{Exchange}$$

$$\frac{\Gamma::x\!:\!\tau\cdot\theta_L::\Delta\vdash M\!:\!\theta_I\rhd K_1 \qquad \tau\cdot\theta_L\multimap\theta_I\leq\tau'\cdot\theta_R\multimap\theta_O\rhd K_2}{\Gamma::\Delta\vdash\lambda(x\!:\!\tau\cdot\theta_L).M\!:\!(\tau'\cdot\theta_R\multimap\theta_O)\rhd K_1\cup K_2}\ \text{Abstraction}$$

$$\frac{\begin{array}{c}\Gamma\vdash M\!:\!(\tau\cdot\theta_L\multimap\theta_I)\rhd K_1\\ \Delta\vdash N\!:\!\theta_R\rhd K_2\end{array} \qquad \Gamma\#\Delta \qquad \begin{array}{c}\tau\cdot\theta_L\multimap\theta_I\leq\tau\cdot\theta_R\multimap\theta_O\rhd K_3\\ \tau\cdot\Delta\leq\Delta'\rhd K_4\end{array}}{\Gamma::\Delta'\vdash MN\!:\!\theta_O\rhd K_1\cup K_2\cup K_3\cup K_4}\ \text{Application}$$

$$\frac{\Gamma\vdash M\!:\!\theta_I\rhd K_1 \qquad \Delta\vdash N\!:\!\theta_I'\rhd K_2 \qquad \Gamma\#\Delta \qquad (\theta_I\otimes\theta_I')\leq(\theta_O\otimes\theta_O')\rhd K_3}{\Gamma::\Delta\vdash M\otimes N\!:\!(\theta_O\otimes\theta_O')\rhd K_1\cup K_2\cup K_3}\ \text{Tensor}$$

$$\frac{\Gamma::x\!:\!\tau_1\cdot\theta_C::y\!:\!\tau_2\cdot\theta_C\vdash M\!:\!\theta_I\rhd K_1 \qquad \theta_I\leq\theta_O\rhd K_2}{\Gamma::x\!:\!\tau\cdot\theta_C\vdash M[x/y]\!:\!\theta_O\rhd K_1\cup K_2\cup\{\tau\geq\tau_1+\tau_2\}}\ \text{Contraction}$$

Figure 6.2.1: An intermediate type system between ICA and Bounded ICA

$$
\begin{array}{rcl}
\mathsf{skip} & : & \mathsf{com} \triangleright \emptyset \\
j_J & : & \mathsf{exp}_J \triangleright \emptyset \\
\mathsf{seq}_{J,\tau} & : & \tau \cdot \mathsf{com} \multimap \tau \cdot \mathsf{exp}_J \multimap \mathsf{exp}_J \triangleright \{\tau \geq \tau\tau, \tau \geq 1\} \\
\mathsf{par}_\tau & : & \tau \cdot \mathsf{com} \multimap \tau \cdot \mathsf{com} \multimap \mathsf{com} \triangleright \{\tau \geq \tau\tau, \tau \geq 1\} \\
\mathsf{if}_\tau & : & \tau \cdot \mathsf{exp}_{\{0,1\}} \multimap \tau \cdot \mathsf{exp}_J \multimap \tau \cdot \mathsf{exp}_J \multimap \mathsf{exp}_J \triangleright \{\tau \geq \tau\tau, \tau \geq 1\} \\
\mathsf{uncurry}_{\tau',\theta_1',\theta_2',\theta,\tau} & : & \tau \cdot (\tau' \cdot \theta_1' \multimap (\tau' \cdot \theta_2' \multimap \theta)) \multimap (\tau' \cdot (\theta_1' \otimes \theta_2') \multimap \theta) \triangleright \{\tau \geq \tau\tau, \tau \geq 1\} \\
\mathsf{op}_{J,\bullet,\tau} & : & \tau \cdot \mathsf{exp}_J \multimap \tau \cdot \mathsf{exp}_J \multimap \mathsf{exp}_J \triangleright \{\tau \geq \tau\tau, \tau \geq 1\} \\
\mathsf{newvar}_{\tau',J,\tau} & : & \tau \cdot (\tau' \cdot (\mathsf{exp}_J \otimes (\tau \cdot \mathsf{exp}_J \multimap \mathsf{com})) \multimap \mathsf{com}) \multimap \mathsf{com} \triangleright \{\tau \geq \tau\tau, \tau \geq 1\}
\end{array}
$$

Figure 6.2.2: Constants of the intermediate type system

which conflate multiple τ are also useful for proving the equivalence in the reverse direction (i.e. that each Bounded ICA term is derivable in the intermediate type system, with solvable constraints).

Another unusual feature of the intermediate type system is that the Subtyping production has effectively been combined into every other production, rather than being separate. This is because ICA does not have a Subtyping rule, and eliminating Subtyping from the type system via this method means that placing Subtyping productions in appropriate places thus becomes part of the Bounded ICA-related proofs, rather than the ICA-related proofs, which is a considerable simplification.

Before can we prove our equivalence, we need to specify the constants for this type system, which are shown in Figure 6.2.2. Just like the type system is Bounded ICA with integer bounds replaced by elements of $\mathbb{A}$ and constraints added, the same transformation is done on the constants. (It should also be noted that $\mathsf{uncurry}$ can be implemented directly in (full) ICA, as $\lambda f.\lambda x.f(\pi_1 x)(\pi_2 x)$, because that language has no restriction on contractions; thus we can make the correspondence between the languages clearer by assuming that ICA has the constant in question.) Of note is the way that the constant bound "1" that appears in the non-base-type constants is represented; we would want to produce a constraint $\tau = 1$, but this is not of one of the four legal forms. Instead, we add constraints $\tau \geq \tau\tau$ and $\tau \geq 1$, which is mathematically equivalent (τ cannot be satisfied by numbers j other than 1 because $j > 1$ implies $j^2 - j = j(j-1) > 0$ because j and $j-1$ are both positive, thus $j^2 > j$ and $j \ngeq j^2$), but which conforms to our requirements for constraints.

We can now prove results about this type system:

Lemma 6.2.5. *If the constraint sets (and side conditions on those constraint sets) were removed from the type system in Figure 6.2.1, then for each tree in the resulting type system, there is exactly one way to add constraint sets to judgements of that derivation such that the result is a derivation in the original type system before constraint sets were removed.*

Proof. It can be observed by structural induction that for any θ_1, θ_2, that there is exactly one K such that $\theta_1 \leq \theta_2 \triangleright K$; that for any τ, Γ_1, Γ_2, there is exactly one K such that $\tau \cdot \Gamma_1 \leq \Gamma_2 \triangleright K$; and by structural induction on the derivation, this proves the lemma. □

Theorem 6.2.6. *An uninferred term M that uses only the constants that exist in Bounded ICA types in ICA-plus-*uncurry *(using the formalization in Figure 5.1.1) if and only if it types in the type system in Figure 6.2.1; additionally, for each derivation of M in either type system, there is a derivation in the other type system with the same number of productions, and the same maximum for the number of base types that appear within any type anywhere in the judgement.*

Proof. We note that the type system of Figure 6.2.1 is identical to the type system of ICA with explicit contractions, except for the presence of a bound τ on every type in a context and the left hand side of every function type, and the existence of the constraint set K. Likewise, constants have the same type in both type systems except for bounds and constraint sets. Thus, we can place an arbitrary bound on each type in a context and on the left hand side of every function type in order to produce a tree that would be an intermediate system derivation if it had constraint sets; and by Lemma 6.2.5, there must be some way to choose constraint sets to produce an intermediate system derivation. □

Lemma 6.2.7. *For any Bounded ICA derivation ∇_1 of an uninferred term M, there is a derivation ∇_4 in the type system in Figure 6.2.1 with root $\Gamma \vdash M, \Theta : \theta \triangleright K$ for which K is solvable (i.e.*

it is possible to consistently replace elements of $\mathbb{A}$ within K with nonnegative integers such that K becomes a set of true statements about integers), ∇_1 contains no more productions than ∇_4, and no type that appears anywhere in ∇_1 contains more occurrences of base types than the type in ∇_4 that contains the most occurrences of base types.

Proof. Delete all uses of the Subtyping rule, and their premises, from ∇_1 to form a new tree ∇_2 (in which for each production whose conclusion was the premise of a Subtyping rule, it now has the judgement that previously was the conclusion of that Subtyping rule as its conclusion.) Choose an arbitrary bijection $g : \mathbb{N} \to \mathbb{A}$ between nonnegative integers and $\mathbb{A}$; and consistently replace every bound j in ∇_2 with $g(j)$ to produce a tree ∇_3. (The idea is to consistently replace each integer with a particular unknown in the constraints, so that we can subsequently solve the constraint system via replacing each unknown with the integer it represents. We thus need g to be a bijection, so that we can later invert it.)

By comparing Figures 6.1.1 and 6.2.1, we observe that ∇_3 would be a derivation in the intermediate system if it had constraint sets (because the rules of the two systems are the same up to bounds, and in uses of the Subtyping rule, the premise and conclusion are the same apart from bounds, so the deletion of the Subtyping rules does not prevent the tree from typing). By Lemma 6.2.5, we can add constraint sets to ∇_3 (in a unique way) to produce an intermediate system derivation ∇_4. It only remains to prove that the constraint set K on the root of ∇_4 is solvable; we already know that ∇_4 is a derivation in the intermediate system, that it contains no more productions than ∇_1, and (because we did not change any parts of the types other than the bounds during the construction of ∇_4, thus they are the same as they were in the original ∇_1) that the types in ∇_1 and ∇_4 are equal up to the replacement of integer bounds with elements of $\mathbb{A}$.

We claim that g^{-1} is such a solution. We note that by structural induction, each constraint in K must have been generated from an Identity rule, a constant, a $\tau \cdot \Gamma \leq \Gamma' \triangleright K$ or $\theta \leq \theta' \triangleright K'$ side condition, or the $\tau \geq \tau_1 + \tau_2$ constraint generated by the Contraction rule (or from multiple such sources). For each such source, we will find that the fact that ∇_1 is a Bounded ICA derivation implies that g^{-1} is a solution to constraints from that source. First, we consider the situation where the production in question corresponds to a single production from ∇_1 (with no intermediate steps deleted due to Subtyping):

- For the Identity rule and for constants, we have constraints $g(1) \geq 1$ and possibly $g(1) \geq g(1) \times g(1)$, which are both solved by g^{-1};
- For the Contraction rule, we have a constraint $g(j + j') \geq g(j) + g(j')$, again obviously solved by g^{-1};
- For constraints generated by $\tau \cdot \Gamma \leq \Gamma' \triangleright K'$ (which only appears in the Application rule), we have $\Gamma' = g(j \cdot \Delta)$, $\tau = g(j)$, $\Gamma = g(\Delta)$, which can easily be shown to be solved by g^{-1} via structural induction;
- For constraints generated by $\theta \leq \theta' \triangleright K'$, we have $\theta = \theta'$, and thus (by structural induction) such constraints are solved by *any* function from $\mathbb{A}$ to $\mathbb{N}$.

The only remaining situation is where intermediate steps were deleted due to the deletion of Subtyping productions; this produces judgements of the form $g(\Gamma) \vdash M', \Theta' : g(\theta') \triangleright K'$ (for $K' \subseteq K$), whereas the above argument shows that the constraints in K' would be satisfied if the same production had instead produced $g(\Gamma) \vdash M', \Theta' : g(\theta) \triangleright K'$ with $\theta \leq \theta'$. An inspection of Figure 6.2.1 shows that the only constraints placed on the type of a term (the θ in $\Gamma \vdash M', \Theta' : \theta \triangleright K$), relative to the premises of productions for which that term type appears in the conclusion, are all of the form $\theta_I \leq \theta \triangleright K''$ for some θ_I. Thus, it is sufficient to prove that if $\theta \leq \theta'$ (in Bounded ICA), then all constraints generated by $g(\theta) \leq g(\theta') \triangleright K''$ are solved by g^{-1}; and this is a straightforward structural induction, comparing the definitions of $- \leq - \triangleright -$ in the intermediate system and $\leq$ in Bounded ICA. □

Lemma 6.2.8. *If there is a derivation ∇_1 of a judgement $\Gamma \vdash M, \Theta : \theta \triangleright K$ in the type system in Figure 6.2.1, and K is solvable on the nonnegative integers, then M has a Bounded ICA derivation.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x\colon 1\cdot\theta_L \vdash x\colon\theta_L}\,\text{Identity}}{x\colon 1\cdot\theta_L \vdash x\colon\theta_R}\,\text{Subtyping}}{x\colon 1\cdot\theta_L :: y\colon(j-1)\cdot\theta_L \vdash x\colon\theta_R}\,\text{Weakening}}{x\colon j\cdot\theta_L \vdash x\colon\theta_R}\,\text{Contraction}$$

$$\dfrac{\dfrac{M\colon\theta_I \text{ is a constant}}{\vdash M\colon\theta_I}\,\text{Constant}}{\vdash M\colon\theta_O}\,\text{Subtyping}$$

$$\dfrac{\dfrac{\Gamma\vdash M\colon\theta_I \qquad x\#\Gamma}{\Gamma :: x\colon j\cdot\theta_W \vdash M\colon\theta_I}\,\text{Weakening}}{\Gamma :: x\colon j\cdot\theta_W \vdash M\colon\theta_O}\,\text{Subtyping}$$

$$\dfrac{\dfrac{\Gamma :: x\colon j'\cdot\theta' :: y\colon j''\cdot\theta'' :: \Delta\vdash M\colon\theta_I}{\Gamma :: y\colon j''\cdot\theta'' :: x\colon j\cdot\theta' :: \Delta\vdash M\colon\theta_I}\,\text{Exchange}}{\Gamma :: y\colon j''\cdot\theta'' :: x\colon j\cdot\theta' :: \Delta\vdash M\colon\theta_O}\,\text{Subtyping}$$

$$\dfrac{\dfrac{\Gamma :: x\colon j\cdot\theta_L :: \Delta\vdash M\colon\theta_I}{\Gamma :: \Delta\vdash \lambda(x\colon\theta_L).M\colon(j\cdot\theta_L \multimap \theta_I)}\,\text{Abstraction}}{\Gamma :: \Delta\vdash \lambda(x\colon\theta_L).M\colon(j'\cdot\theta_R \multimap \theta_O)}\,\text{Subtyping}$$

$$\dfrac{\dfrac{\dfrac{\Gamma\vdash M\colon(j\cdot\theta' \multimap \theta_I) \qquad \Gamma\#\Delta \qquad \Delta\vdash N\colon\theta'}{\Gamma :: j\cdot\Delta\vdash MN\colon\theta_I}\,\text{Application}}{\Gamma :: \Delta'\vdash MN\colon\theta_I}\,\text{(see text)}}{\Gamma :: \Delta'\vdash MN\colon\theta_O}\,\text{Subtyping}$$

$$\dfrac{\dfrac{\Gamma\vdash M\colon\theta_I \qquad \Delta\vdash N\colon\theta_I' \qquad \Gamma\#\Delta}{\Gamma :: \Delta\vdash M\otimes N\colon(\theta_I\otimes\theta_I')}\,\text{Tensor}}{\Gamma :: \Delta\vdash M\otimes N\colon(\theta_O\otimes\theta_O')}\,\text{Subtyping}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma :: x\colon j_1\cdot\theta_C :: y\colon j_2\cdot\theta_C\vdash M\colon\theta_I}{\Gamma :: x\colon(j_1+j_2)\cdot\theta_C\vdash M[x/y]\colon\theta_I}\,\text{Contraction}}{\Gamma :: x\colon(j_1+j_2)\cdot\theta_C :: y\colon(j-j_1-j_2)\cdot\theta_C\vdash M[x/y]\colon\theta_I}\,\text{Weakening}}{\Gamma :: x\colon j\cdot\theta_C\vdash M[x/y]\colon\theta_I}\,\text{Contraction}}{\Gamma :: x\colon j\cdot\theta_C\vdash M[x/y]\colon\theta_O}\,\text{Subtyping}$$

Figure 6.2.3: Admissibility of the intermediate type system

Proof. Imagine a type system that is the same as the intermediate type system, except that it uses integers rather than elements of $\mathbb{A}$, and the K instead become side conditions on the values of those integers. We can remove the constraint sets K from ∇_1, and replace each $\tau \in \mathbb{A}$ with the integer that maps to τ in the solution of K, to form a derivation ∇_2 of M in this new type system. It thus suffices to show that all the rules of this new type system are admissible in Bounded ICA (i.e. that they are equivalent to some sequence of existing Bounded ICA rules). We show admissibility via explicitly giving the equivalent Bounded ICA derivations for each rule; these are shown in Figure 6.2.3. We note that the side conditions on each Subtyping rule in Bounded ICA are implied by the K in the intermediate type system, by structural induction; and the $j-1$ that appear in the expansion of Identity and $j-j_1-j_2$ that appear in the expansion of Contraction are nonnegative integers because we have $j \geq 1$ and $j \geq j_1 + j_2$ respectively.

It remains to show that if $\tau \cdot \Delta \leq \Delta' \triangleright K$, then after replacing each τ with the corresponding j in a solution to K, $\Gamma :: \Delta' \vdash MN : \theta_I$ can be derived from $\Gamma :: j \cdot \Delta \vdash MN : \theta_I$ in Bounded ICA. We note that by definition, the only difference between $j \cdot \Delta$ and Δ' is that Δ' can have higher bounds on variables (but never lower bounds). Therefore, for each variable that appears in Δ, we can use Weakening to add a fresh variable of the same type, which has a bound equal to the difference in bounds between the original variable's appearance in Δ' and in $j \cdot \Delta$ (this difference must be non-negative). Then we insert Contraction and Exchange rules in the obvious manner to contract these fresh variables with the corresponding original variables. □

Theorem 6.2.9. *If for all constraint sets K, it is decidable whether there is a solution to that constraint set, then for all uninferred terms M, it is decidable whether those terms have a Bounded ICA type.*

Proof. We give the following algorithm:

1. Calculate j and k for M, as defined in Theorems 6.2.3 and 6.2.4 respectively.

2. Calculate the maximum number l of elements of $\mathbb{A}$ that can appear in an intermediate system derivation that uses no more than j productions and no more than k base types within

any single type. (This maximum exists, and can be calculated in finite time, because if all elements of $\mathbb{A}$ are considered equivalent, which is enough to perform the calculation, then with the limit k on the complexity of types there are only a finite number of axioms and only a finite number of ways to apply inference rules in the type system, and so it is possible simply to try all possibilities up to the limit j of productions.)

3. Choose an arbitrary finite subset $A \subset \mathbb{A}$ that has at least l elements.

4. Find all derivations for M in the intermediate type system that use no more than j productions, no more than k base types within any single type, and no elements of $\mathbb{A}$ other than those in A. (There are finitely many such derivations, and they can be found in finite time, for the same reason as step 2).

5. For each such derivation, let its root be $\Gamma \vdash M, \Theta : \theta \triangleright K$. If any such K is solvable, then M has a Bounded ICA type. If no such K is solvable, then M does not type in Bounded ICA.

We need to show that this algorithm always produces the correct result, and that it always terminates. Termination is easier; given our assumption, determining whether a constraint set is solvable is decidable (and thus terminates), and all the other steps have been shown to take finite time. For correctness, we have two cases:

- If M has a Bounded ICA type, then by Theorems 6.2.3 and 6.2.4, it has a Bounded ICA derivation within the bounds set by j and k; and thus by Lemma 6.2.7, M has a derivation ∇ in the intermediate type system that obeys the same bounds j and k, and whose root has a solvable constraint set. By the definition of l, ∇ contains no more than l elements of $\mathbb{A}$. Because which elements of $\mathbb{A}$ are used in a derivation does not matter (merely which elements are equal or unequal), without loss of generality, ∇ contains no elements of $\mathbb{A}$ other than those in A. Thus, ∇ was found in step 4; and because its root has a solvable constraint set, step 5 reported that M had a Bounded ICA type.

- If M does not have a Bounded ICA type, then step 5 cannot report that it does have a Bounded ICA type; if it did, then there would have to be some derivation for M in the intermediate type system whose root had a solvable constraint set, meaning that by Lemma 6.2.8, M would have a Bounded ICA type, a contradiction.

□

The remaining step is to show that for all sets of constraints that obey the grammar for κ, it is decidable whether those sets are solvable. This is far from obvious; such constraints include inequalities, additions, and multiplications, which unrestricted is enough power to produce arbitrary Diophantine equations, which are famous for being undecidable (this is Hilbert's Tenth Problem, which was proved undecidable by Matiyasevič; Davis presents this result in English in [10]). (A problem is "decidable" if an algorithm exists that always either finds a proof that that problem is solvable, or else produces a proof that that problem is unsolvable.) This is why we limited all our constraints to one of four specific forms: to make the resulting constraint system decidable:

Theorem 6.2.10. *Given any set K of constraints κ of the forms $j_k \geq j_l$, $j_k \geq l$, $j_k \geq j_l + j_m$, and $j_k \geq j_l j_m$, it is a decidable problem to either find some assignment of nonnegative integers to variables such that all constraints are satisfied, or else to prove that no such assignment exists.*

Proof. We prove decidability via giving an explicit algorithm:

1. First, we solve the constraint system, not in $\mathbb{N}$, but in a different semiring: instead of taking each j_k to be a nonnegative integer, we take each j_k to be an element of the commutative semiring with elements $\{\mathbf{0}, \mathbf{1}, \infty\}$, with addition defined as $\mathbf{0} \oplus j = j$, $\infty \oplus j = \infty$, $\mathbf{1} \oplus \mathbf{1} = \infty$, and multiplication defined as $\mathbf{0} \odot j = \mathbf{0}$, $\mathbf{1} \odot j = j$, $\infty \odot \infty = \infty$. Because there are only finitely many assignments of values to variables in this semiring, it is possible to discover all solutions in this semiring merely via trying all possibilities. If there are no possibilities, then the constraint system on integers is unsolvable. Otherwise, the fol-

lowing steps are repeated for each solution that was found in the $\{\mathbf{0},\mathbf{1},\infty\}$ semiring until some solution on nonnegative integers is found.

2. For each variable found to be **0** in the semiring above, assign 0 as the value of that variable, and replace all occurrences of that variable with 0 in the constraints. Likewise for **1** and 1. For each variable found to be ∞ in the semiring above, add a constraint that that variable is ≥ 2. Replace any expressions of the form $k+l$ or kl on the right hand side of constraints for which k and l are constants with their numerical values.

3. Delete all constraints of the forms $j_k \geq 0j_l$, $j_k \geq j_l \times 0$, $j_k \geq 0$, and $j_k \geq 1$. Simplify all constraints of the forms $j_k \geq j_l + 0$, $j_k \geq 0 + j_l$, $j_k \geq 1j_l$, and $j_k \geq j_l \times 1$ to $j_k \geq j_l$.

4. If there are any constraints remaining of the form $j_k \geq j_k + j_l$ or $j_k \geq j_k j_l$, then the constraint system is unsolvable for this particular solution in the $\{\mathbf{0},\mathbf{1},\infty\}$ semiring: continue with checking the other solutions. If the constraint system is unsolvable for all solutions in the $\{\mathbf{0},\mathbf{1},\infty\}$ semiring, it is unsolvable on integers as well.

5. Construct the relation $\succeq$ as the minimum transitive relation such that $j_k \geq j_l$ implies $j_k \succeq j_l$, and $j_k \geq j_l + j_m$ and $j_k \geq j_l j_m$ each imply $j_k \succeq j_l$ and $j_k \succeq j_m$.

6. If there are no k, l with $k \neq l$ such that $j_k \succeq j_l$ and $j_l \succeq j_k$, skip forwards to step 7. Otherwise, pick some such k and l, and consistently replace j_k with j_l in every constraint; if and when a value is eventually assigned to j_l in step 7, assign the same value to j_k. Then go back to step 4.

7. Pick some k such that there is no $l \neq k$ such that $j_k \succeq j_l$. (Some such k must exist; otherwise, $\succeq$ would be a well-founded relation on finitely many values with no minimum, which is impossible.) Thus, all constraints with j_k on the left hand side are of the form $j_k \geq l$ for some constant l (and due to the constraints added in step 2, there must be at least one such constraint). Choose the largest value of l that appears in any of these constraints, and assign that value to j_k. Then consistently replace j_k with l in every

constraint, remove all constraints of the form $m \geq n$ (where m and n are constants), and replace any expressions of the form $m+n$ or mn on the right hand side of constraints (where m and n are constants) with their numerical values. Repeat this step until all j_k have a value assigned; the resulting assignment is a set of solutions to the constraints.

This algorithm must terminate: every time the algorithm returns to step 4, it is immediately after deleting all instances of a specific variable from the constraint system, and likewise each repeat of step 7 deletes all instances of a specific variable from the constraint system. Thus, no step can run more times than the number of solutions in the $\{\mathbf{0}, \mathbf{1}, \infty\}$ semiring times the number of variables in the constraint system, meaning that the algorithm always terminates.

If the algorithm finds no solution, then there can be no solution. For any solution (on nonnegative integers) to the constraint system, there is a solution on the $\{\mathbf{0}, \mathbf{1}, \infty\}$ semiring formed via replacing 0 with $\mathbf{0}$, 1 with $\mathbf{1}$, and all higher integers with ∞; and none of the other steps will transform the constraint system in such a way that the constraints given become false. Thus, step 4 could never report that the solution on the semiring fails to generalize to the integers as a whole, because doing so would imply $j_k \geq j_k + j_l$ or $j_k \geq j_k j_l$ with $j_k \geq 2$ and $j_l \geq 2$, which is mathematically impossible; and thus the algorithm cannot report failure, and (because it always terminates) must report success.

If the algorithm finds a solution, then it must be correct. None of the deletions of constraints in step 3 remove any requirements from the constraint system as a whole, because there is a requirement $j_k \geq 2$ on every variable that exists in the constraint system at that point (added in step 2, and never deleted). Likewise, none of the deletions of constraints in step 7 remove any requirements from the constraint system as a whole, because the only constraints of the form $m \geq n$ are those produced via replacing $j_k \geq l$ with $m \geq l$ (where m is the largest such value of l), and so all the removed constraints are tautologies. All other manipulations of constraints replace them with mathematically equivalent constraints.

Thus, the algorithm always terminates, and produces a correct solution to the constraint system if and only if there is such a solution; and thus, the constraint system is decidable. $\square$

Corollary 6.2.11. *Type inference for Bounded ICA is decidable (i.e. an algorithm exists that, for any uninferred Bounded ICA term, either finds a proof that a derivation of that term exists, or finds a proof that no such derivation exists).*

There is actually a stronger result than decidability for Bounded ICA here, as can be seen by examining the proofs of the theorems. All the decidability proofs are constructive (i.e. when a term is proved to be derivable, the proof that is given is an explicit derivation tree or an explicit solution to the constraint system), meaning that this section actually gives a sound and complete (if staggeringly inefficient) algorithm for Bounded ICA inference: we can determine not just whether the term types in Bounded ICA, but also a derivation tree, complete with types.

An obvious question is whether this algorithm can be improved into something that is practically usable in a compiler. The answer appears to be yes; the compiler discussed in Chapter 10 currently does SCC inference as one of the stages of compilation, and it does so using effectively the algorithm discussed in this section. However, for efficiency, instead of iterating over all possibilities, it merely chooses the "minimum" possibility (in some appropriate sense), each time, which is much faster. This algorithm is clearly sound, in that if it finds an SCC derivation, it will follow the rules of SCC. This suggests a conjecture that this cut-down version algorithm is also complete: if an SCC derivation exists, even this algorithm will be capable of finding some such derivation. This conjecture is still an open problem, which may be an interesting topic for further research.

6.2.1 Negative results

In scientific research, it is important to report your failures as well as your successes; picking only the experiments that gave the results you wanted can give an incorrect picture of the evidence, leading perhaps to something seeming true when the evidence does not support it. In more mathematical fields like the one in this thesis, there is less of a risk of this; barring mistakes in a proof, a result that is proved correct is correct no matter how much effort was spent failing to prove it. Nonetheless, it is worth cataloging plausible-looking conjectures that nonetheless turned out to be false; this may help explain some of the decisions I make (the alternatives may well have not worked), and will hopefully reduce duplicated effort in trying to prove false statements.

These results do not need much elaboration; as opposed to positive results, a single counterexample is enough to produce a negative result. Thus, this section is just a list of counterexamples, without proofs that the example actually does satisfy the statement made about it. (You can negate the facts to produce the original conjectures I was trying to prove.)

Fact 6.2.12. *There are ICA terms such that cycles exist in the constraints produced for those terms via the intermediate type system in Figure 6.2.1, in the sense that the $\succeq$ relation defined in Theorem 6.2.10 is not well-founded (i.e. step 6 is not redundant).*

Almost any term with no constants but skip that does not type in SCC will serve as an example for this, e.g. $(\lambda g.g(\lambda x.g(\lambda y.x)))(\lambda f.f(f(\mathsf{skip})))$ (given by Ghica, Murawski and Ong in [18]). There are examples that do type in SCC, too, such as the counterexample for the next conjecture.

Fact 6.2.13. *There are terms which type in SCC and Bounded ICA, and which have all types for that term differ only in the bounds, but for which there is a covariant position in those types such that the set of bounds that can appear in that position in some type for that term is not the same as the set of all nonnegative integers.*[2]

This result is perhaps surprising, as the bounds in each covariant position can range over all nonnegative integers in almost all small example terms. Although it is possible that simpler counterexamples exist, the smallest I am aware of is:

$$\lambda q.(\lambda g.g(\lambda x.g(qx)))(\lambda b.(\lambda k.((k(\lambda u.u))(\lambda l.((kb)(\lambda m.(l(m(\mathsf{skip}))))))))(\lambda v.\lambda w.wv))$$

This term has type $j^2 \cdot (1 \cdot \mathsf{com} \multimap j \cdot \mathsf{com} \multimap \mathsf{com}) \multimap \mathsf{com}$. The bound 1 on the type could be a 0 instead, but cannot be any value greater than 1 (and thus ranges over the set $\{0,1\}$ rather

[2]The negation of this was conjectured by Dan Ghica, the counterexample was found by me; I previously published this counterexample in [20].

than $\mathbb{N}$). This conjecture may be almost correct; I conjecture that if a term can be typed in SCC or Bounded ICA with a bound of 2 or more in a covariant position, then it allows all nonnegative integers in that position.

Fact 6.2.14. *There are closed terms which have arbitrarily large ICA (with explicit contractions), SCC, and Bounded ICA derivations, no two judgements within which are equal.*

This is the reason why the proofs in this section needed to go to so much effort to place limits on the complexity of derivations, rather than merely enumerate all derivations (or all derivations that contained no repeated judgements). Such terms can be very simple, such as $\lambda y.\lambda x.x$; the actual derivation is a little less obvious, but basically operates via starting with Identity on a variable (x in this example), introducing arbitrarily many new variables (with bound 0, in those type systems that use bounds) via Weakening, Abstracting x, Contracting all the remaining variables into one (y in this example), then finally Abstracting the remaining variable.

6.3 The polymorphism problem

In Section 6.1, we saw the technique of introducing bounds on function application in order to make function equivalence in ICA decidable, observing that the undecidability of ICA stems from the lack of bounds on free variables and on the existence of the fix constant. This immediately suggests a conjecture:

Conjecture 6.3.1. *All closed ICA terms that do not use the* fix *constant anywhere type in Bounded ICA.*

Sadly, however, this conjecture is false. Ghica, Murawski and Ong give the following counterexample in [18]:

$$(\lambda g.g(\lambda x.g(\lambda y.x)))(\lambda f.f(f(\mathsf{skip})))$$

The authors of the above paper proved that this term has no SCC type (and thus no Bounded ICA type, because as it does not use products or tensors anywhere, any type derivation for it

$$\theta_1 \triangleq \mathsf{com}$$
$$\theta_j \triangleq \theta_{j-1} \to \mathsf{com}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{g:\theta_3 \vdash g:\theta_3} \quad \dfrac{\dfrac{\dfrac{}{g':\theta_3 \vdash g':\theta_3} \quad \dfrac{\dfrac{\dfrac{}{x:\theta_1 \vdash x:\theta_1}}{x:\theta_1 :: y:\theta_1 \vdash x:\theta_1}}{x:\theta_1 \vdash \lambda y.x:\theta_2}}{g':\theta_3 :: x:\theta_1 \vdash g'(\lambda y.x):\theta_1}}{g':\theta_3 \vdash \lambda x.g'(\lambda y.x):\theta_2}}{g:\theta_3 :: g':\theta_3 \vdash g(\lambda x.g'(\lambda y.x)):\theta_1}}{\dfrac{g:\theta_3 \vdash g(\lambda x.g(\lambda y.x)):\theta_1}{\vdash \lambda g.g(\lambda x.g(\lambda y.x)):\theta_4}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{f:\theta_2 \vdash f:\theta_2} \quad \dfrac{\dfrac{}{f':\theta_2 \vdash f':\theta_2} \quad \dfrac{}{\vdash \mathsf{skip}:\theta_1}}{f':\theta_2 \vdash f'(\mathsf{skip}):\theta_1}}{f:\theta_2 :: f':\theta_2 \vdash f(f'(\mathsf{skip})):\theta_1}}{f:\theta_2 \vdash f(f(\mathsf{skip})):\theta_1}}{\vdash \lambda f.f(f(\mathsf{skip})):\theta_3}}{\vdash (\lambda g.g(\lambda x.g(\lambda y.x)))(\lambda f.f(f(\mathsf{skip}))):\theta_1}$$

Figure 6.3.1: ICA derivation of $(\lambda g.g(\lambda x.g(\lambda y.x)))(\lambda f.f(f(\mathsf{skip})))$

$$\begin{aligned}
&(\lambda g_7.\lambda g_4.\lambda g_1.\lambda g_0.g_0(\lambda x_3.\lambda x_2.g_1(\lambda y.x_2)(\lambda y.x_3))\\
&\qquad\qquad\qquad\qquad\quad (\lambda x_6.\lambda x_5.g_4(\lambda y.x_5)(\lambda y.x_6))\\
&\qquad\qquad\qquad\qquad\quad (\lambda x_9.\lambda x_8.g_7(\lambda y.x_8)(\lambda y.x_9)))\\
&(\lambda f_{11}.\lambda f_{10}.f_{10}(f_{11}(\mathsf{skip})))\\
&(\lambda f_{13}.\lambda f_{12}.f_{12}(f_{13}(\mathsf{skip})))\\
&(\lambda f_{15}.\lambda f_{14}.f_{14}(f_{15}(\mathsf{skip})))\\
&(\lambda f_{18}.\lambda f_{17}.\lambda f_{16}.f_{16}(f_{17}(\mathsf{skip}))(f_{18}(\mathsf{skip})))
\end{aligned}$$

Figure 6.3.2: Serialization of $(\lambda g.g(\lambda x.g(\lambda y.x)))(\lambda f.f(f(\mathsf{skip})))$

would exist entirely in the common subset of Bounded ICA and SCC). The term, however, can be typed in full ICA, as shown in Figure 6.3.1. The obvious next question is as to whether the term "uses unbounded contraction", in some sense; although the question is ill-defined, there is considerable evidence that suggests we should define the question such that the answer is no. One argument is that in most reasonable semantics, the term is equivalent to skip, with finitely many steps in the evaluation and no possibility of divergence. Later in this chapter, in Section 6.5, we consider ways of transforming terms that use bounded contraction into terms that use no contraction while keeping an equivalent semantics ("linearization" in Bounded ICA-like type systems, "serialization" in SCC-like type systems); this provides an obvious alternative definition of unbounded contraction (terms for which such a transformation is impossible). On this particular term, it is impossible to use Bounded ICA or SCC to help guide this transformation; however, algorithms for performing it exist that work directly with ICA (I presented such an algorithm in my Master's thesis, [44], although that algorithm is not known to work in every case), or even with untyped lambda calculus (such as the algorithm by Kfoury in [28]). And these algorithms have no problems with this term; Kfoury's linearization can handle any β-strongly normalizing term ([28, Theorem 3.15 on page 19]), and my algorithm also produces a correct serialization of the term (shown in Figure 6.3.2).

We now have answers to some of the questions in the preceding section: it is decidable whether a term has a Bounded ICA or SCC type, but this is not equivalent to using only bounded contraction in general. SCC does its job in one respect – it presents a reasonably large subset of ICA which has decidable equivalence – but fails in another, in that it fails to fully capture the set of terms that use only bounded contraction. Bounded ICA was intentionally designed to explore the same issues as appear in SCC, and thus it (intentionally) has the same issues.

Of course, this just raises more questions. One of the more obvious is as to whether non-generally-recursive ICA terms that fail to type in SCC have to be contrived terms such as $(\lambda g.g(\lambda x.g(\lambda y.x)))(\lambda f.f(f(\mathsf{skip})))$, or whether they can appear more naturally in programming; this question is comparatively minor theoretically, but important to judging the usefulness of SCC for some practical compiler project, and is answered later in this section. If SCC cannot be used, what about some other type system; is there a type system which can type all ICA terms that use only bounded contractions? I consider that question in Section 6.4. Both these questions, though, are easier to answer after first considering a much more fundamental question: why are SCC and Bounded ICA less general than might be expected?

The answer is even more general and fundamental than the question; the fault is not with SCC or Bounded ICA in particular, but with the idea of resource-bounding type systems in general, whether those resources are contraction bounds (as in this case), or something entirely different. The problem is that Bounded ICA – and all such type systems – capture more details of the term than ICA (or whatever unbounded type system they are based on) do, i.e. two terms with the same ICA type can have different Bounded ICA types. Thus, a function that is monomorphic in ICA (i.e. needing to operate only on one type of argument) can be polymorphic in Bounded ICA, operating on arguments whose types, despite being similar (as they are based on the same ICA type, they are equal up to bounds), are different; and because neither Bounded ICA nor any of the other type systems we look at in this thesis support polymorphism, a term that requires polymorphism effectively cannot be typed at all.

It is possible to express this fact as a general theorem about type systems:

Theorem 6.3.2. *If a type system S can type every term in simply typed lambda calculus, and it can only contract two terms if they have the same type (in S), and if for all terms $M(N)$, the* *type of N is entirely determined by the type of M, then for any two terms with the same type in simply typed lambda calculus, there is a type in S that both those terms have.*

Proof. Let the two terms be M and N. We can construct a term in simply typed lambda calculus that gives both M and N as arguments to the same function f (there are numerous examples of this, e.g. $M,N \vdash (\lambda f.f(M)(f(N)(\lambda z.z)))(\lambda x.(\lambda y.y))$ which is correctly typed in simply typed lambda calculus so long as M and N have the same type). When this term is typed in S, the two uses of f must have the same type (because they are contracted, and we assumed that terms being contracted in S have the same type), and thus their arguments must have the same type, i.e. M and N have the same type. □

This theorem illustrates a very important point about type systems like Bounded ICA whose purpose is to *describe* a term via producing a type for it that conveys more information than its simply typed lambda calculus type would (e.g. in the case of Bounded ICA, via specifying the contraction bounds on each variable), rather than type systems whose purpose is to *disallow* terms that fail to meet certain conditions (a more common use of type systems): if such type systems are to be able to describe any term, then either they have types general enough that any two terms with the same type in typed lambda calculus can have the same type in the type system, or they have an "unusual" application rule (in that $M(N)$ allows multiple different types for N even if the type of M stays the same), or they have an "unusual" contraction rule (in that two terms can be contracted even if they have different types).

It is thus instructive to consider possible ways in which the premises of Theorem 6.3.2 could fail to hold, or in which its conclusion might be less disastrous. One approach is to add particularly general types to the type system; the idea is that if a function needs to be able to handle terms of two different types, it should just be given a type which accepts the most general argument that the function might need. This is the approach that SCC took in an attempt to work around the fact that its types are more expressive than those of simply typed lambda calculus; it has a complex Subtyping rule that, in many simple cases, allows a common type to be found for two terms that act quite differently.

There are many type systems for which this approach is all that is necessary. Consider, as an example, a type system that is like simply typed lambda calculus, except that it distinguishes "pure" terms from terms that are not necessarily pure (for some appropriate definition of "pure"). We can note that for most applications of this sort of type system, there is no problem with treating pure terms as potentially impure, to allow them to be contracted with terms that are actually impure. In type systems that genuinely do have a "most general" version of a type, there is no particular problem expressing all terms of typed lambda calculus in that type system; in the degenerate case, all terms can just be given their most general type.

This is not true polymorphism, though; although it can work well on base types, it is unable to express any sort of relationship between a function's argument, and its return value. A function that can make few assumptions about its argument will not be able to make many guarantees about its return value, and in particular, the guarantees it makes about the return value cannot depend on the assumptions the context allows it to make about the type of the argument. In the case of SCC, then, it is not surprising that this mechanism fails to capture the behaviour of all ICA terms, even when general recursion is excluded.

Knowing the reason that not all non-generally-recursive ICA terms type in SCC, it also becomes clear that not all such terms are pathological. For example, I constructed a reasonable-looking program in Verity (described in Chapter 10), shown in Figure 6.3.3; the idea of the program is to run an algorithm (given as a free variable `matrixinit`; the program is parametric on it) on a range of inputs (ranging from $\langle 0,0\rangle$ to $\langle 9,9\rangle$ in this example) and record the outputs in a heap-allocated matrix. None of this behaviour is particularly unreasonable; and the implementation of the benchmark program does not seem unreasonable either (the highest-order function in the example is the second-order `foreach` which basically implements a `map` operation on dynamically allocated arrays, at least if you consider var to be a base type rather than a tensor that contains a function type). The current Verity compiler operates via finding an SCC type for the Verity program using essentially the algorithm in Section 6.1 (with some optimizations so that it does not require exponential time to run, although I have not proved

```
# A memory that pointers can point into, C-style
# 64K should be enough for anyone
new heap(65536) in

# A very simple memory allocator
new brk := 0 in
let malloc = \size. (brk := !brk + size; !brk - size) in

# Iterator over arrays on the heap
let foreach = \base.\count.\func. {
    new index := 0 in
    func(!index, heap(base));
    while !index < count do {
        index := !index + 1;
        func(!index, heap(base + !index))
    }
} in

# Allocate a 10x10 matrix as an array-of-arrays
new aoa := malloc(10) in
foreach !aoa 10 (\(y,p). p := malloc(10));

# Initialize the matrix according to a given algorithm
foreach !aoa 10
    (\(y,a). foreach !a 10
        (\(x,e). e := matrixinit(x,y)))
```

Figure 6.3.3: A Verity program with an ICA type but no SCC type

Glasgow Haskell Compilation System v7.6.3:

```
\m -> \n -> (\f -> f(m)(f(n)(\z -> z)))(\x -> (\y -> y))
  :: t -> t -> t1 -> t1
```

Objective Caml toplevel v3.12.1:

```
# fun m -> fun n -> (fun f -> f(m)(f(n)(fun z -> z)))
                    (fun x -> (fun y -> y)) ;;
- : 'a -> 'a -> 'b -> 'b = <fun>
```

F# (fsharp-2.0.0): (this uses an intentional type error so that the compiler will print types)

```
(fun m -> fun n ->
 (fun f -> f(m)(f(n)(fun z -> z)))(fun x -> (fun y -> y)))
+ (4:int)
error FS0001: The type 'int' does not match the type
''a -> 'a -> 'b -> 'b'
```

gosc Verity compiler v0.09.15.3:

```
m:?30, n:?30 |-
    ((\f.((f(m))((f(n))(\z.z))))(\x.(\y.y))) : (?38 -> ?38)
```

Glasgow Haskell Compilation System with -XRankNTypes -XScopedTypeVariables:

```
\m -> \n -> (\(f :: forall a b. a -> (b -> b)) ->
               f(m)(f(n)(\z -> z)))(\x -> (\y -> y))
  :: a -> a1 -> t -> t
```

Figure 6.3.4: Type inference for $(\lambda f.f(M)(f(N)(\lambda z.z)))(\lambda x.(\lambda y.y))$

that these optimizations preserve completeness); and it states that there is no SCC type for the program. This example is also quite striking in that it uses at most second-order functions (the fact that it uses var is not essential to the failure to type), and no concurrency; I actually discovered it by accident while trying to prove a conjecture that such counterexamples require at least third-order functions. It seems plausible that this sort of problem will eventually occur in any sufficiently large program that is written in a functional style.

Thus, it seems safe to conclude that although subtyping rules can help ameliorate the problem of creating a descriptive type system, they require sacrificing too much expressive power to solve the problem on their own; and as such, we would desire an application or contraction rule more complex than that of typed lambda calculus. One possibility is simply to add polymorphism to the language (thus generalizing the Application rule); it is no problem if some terms become unexpectedly polymorphic if the language admits polymorphism anyway. The main issue with this seems to be that type inference becomes much more complex, beyond the abilities of practical type inference algorithms nowadays, because terms need to be given types that are not the most "obvious". Although the premises of Theorem 6.3.2 fail to hold in a polymorphic setting, the result still holds so long as the only types of $(\lambda f.f(M)(f(N)(\lambda z.z)))(\lambda x.(\lambda y.y))$ are of the form $M:\theta', N:\theta' \vdash \theta \to \theta$ (it would need a type of $M:\theta', N:\theta'' \vdash \theta \to \theta$ to defeat the result).

We can try some of today's practical type inference algorithms on this term to see what types they give; the result is shown in Figure 6.3.4 (which aims to use the most lambda-calculus-like syntax those implementations will accept, rather than idiomatic syntax for the language, to make comparisons easier).[3] The output of Verity's compiler `gosc`, which is aiming to make practical use of the research in this thesis, is also shown; although it does not support polymorphism, its type inference algorithm does (it first infers a type and then complains if it is not monomorphic, and thus can infer polymorphic types even though it cannot compile them), but it cannot distinguish between M and N either.

What about the prospects for the future? As the last example shows, at least one implementation can handle the type $M:\theta', N:\theta'' \vdash \theta \to \theta$ for the term, even though it requires enabling two extensions that are not yet standard Haskell, and giving an explicit type annotation because the inference algorithm cannot infer the type on its own.[4] Regular lambda calculus cannot handle such types, but second-order lambda calculus (also known as System F) can. (System F was discovered by Reynolds in [39]; Girard apparently independently discovered the same type system in a thesis two years earlier, but I have not been able to find an uncorrupted copy of this thesis to verify this.) The Haskell implementation is using a type system that embeds System F

[3]Thanks to John Berry for help with the F# syntax used here.

[4]Thanks to Shachaf Ben-Kiki, who told me that this type annotation could be used to give the term this type.

in order to handle typing the term here.

It is not all good news: type inference for System F is in general undecidable, as Wells showed in [50], and so it is not surprising that the Haskell program required a type annotation in order to give the required type. However, the term in question is contained within the "rank-2" fragment of System F, which does have decidable type inference (as Kfoury and Tiuryn show in [29]), although type inference for the rank-2 fragment does not yet seem to have been implemented in any mainstream language. It is thus unclear to me whether polymorphism can potentially serve as a practically useful solution to the problem illustrated by Theorem 6.3.2; if rank-2 polymorphism is enough, then it can, but there may be more complicated examples for which rank-2 polymorphism is insufficient.

Another potential approach to the problem illustrated by Theorem 6.3.2 is the use of dependent typing. I am aware of ongoing work by other groups in this direction (in addition to the occasional published paper, such as [14] by Gaboardi et al); it is particularly appropriate when the languages in question are dependently typed anyway. The main drawback is that the addition of dependent typing is quite a major change to a language; a simpler solution would be desirable in order to avoid the need to adapt existing results to a dependently-typed setting.

One final approach I would like to consider is that of intersection types (which have been discovered independently multiple times, e.g. by Coppo and Dezani-Ciancaglini in [9]); this can either be seen as a change to the contraction behaviour (allowing the contracting together two terms of different type), or as similar to a subtyping rule (if a term has types θ and θ', it also has type $\theta \cap \theta'$). These are probably inappropriate as a general-purpose solution to the problem, because they have a tendency to lead to undecidable type systems; the paper by Coppo and Dezani-Ciancaglini proves that terms are normalizable if they can be given an intersection type, and a result [38] by Pottinger proves that terms can be given an intersection type if they are normalizable, producing both directions of an "if and only if" implication. Normalizability is quite a similar concept to halting, which is undecidable in general. However, in the context of a finite-state semantics, this is less of a problem, because the halting problem is decidable simply by running the program to see if it repeats a previous state or terminates (one or the other must happen eventually); this means that a side-condition designed to ensure that a program is finite-state may also be sufficient to make its type inference decidable. In the case of SCC and Bounded ICA, intersection types thus turn out to be possibly the best available solution, as is explained in the next section.

6.4 Bounded intersection types

As we have seen, the major issue with Bounded ICA is that it fails to type some terms that we would expect it to type, and the reason is that it cannot contract together two terms that have different Bounded ICA types (but the same ICA type). Thus, one way to fix this problem would be to create a variant of Bounded ICA that permits such contractions. The result is a system of *bounded intersection types*. The main difference from normal intersection types is that normally, $x:\theta \cap \theta'$ means "the variable x can be used either with the type θ, or with the type θ'"; in bounded intersection types, we use the notation $x:\theta + \theta'$ for "the variable x is used twice, once with type θ, and once with type θ'". Thus, a Bounded ICA type such as $5 \cdot \theta$ can be seen as $\theta + \theta + \theta + \theta + \theta$, leaving bounded intersection types as a generalization of bounded types (and as a special case of intersection types, in that they place more restrictions on the term).

It is worth first considering the prior work on the subject, by Kfoury in [28]. Kfoury was mostly interested in untyped lambda calculus, and in particular, which untyped lambda calculus terms were β-normalizing in various senses (a very similar notion to that of program termination). Part of this work was the introduction of a type system $\boldsymbol{\lambda}$, shown in Figure 6.4.1 (in the notation used in this thesis, rather than Kfoury's original notation).

We would like to use Kfoury's type system as a basis for a generalization of Bounded ICA, but several changes are needed. The first thing to note is that Kfoury merged several rules which are normally kept separate; "K-Abstraction" is a merge of the normal Abstraction and Weakening rules, and the Application rule is incredibly complex (when the subsidiary definition $\Gamma + \Delta$ is

(Adapted from [28, page 21].)

$$\sigma ::= \mathsf{val} \text{ (i.e. there is only one base type)}$$
$$\gamma ::= \sigma \mid \theta \multimap \gamma$$
$$\theta ::= \gamma \mid \theta + \theta$$
$$\Gamma ::= \text{set of } x\colon\theta$$

$$\begin{aligned}\Gamma + \Delta \triangleq \{x\colon\theta \mid {} & (x\colon\theta \in \Gamma \wedge \nexists \theta'.x\colon\theta' \in \Delta) \vee \\ & (x\colon\theta \in \Delta \wedge \nexists \theta'.x\colon\theta' \in \Gamma) \vee \\ & (\theta = \theta_1 + \theta_2 \wedge x\colon\theta_1 \in \Gamma \wedge x\colon\theta_2 \in \Delta)\}\end{aligned}$$

$$\frac{x \neq y}{x\colon\theta \# y\colon\theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma :: \Gamma'\#\Delta} \qquad \frac{x\colon\theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{x\colon\gamma \vdash x\colon\gamma}\text{ Identity}$$

$$\frac{\Gamma \cup \{x\colon\theta\} \vdash M\colon\gamma \qquad x\#\Gamma}{\Gamma \vdash \lambda(x\colon\theta).M\colon\theta \multimap \gamma}\ I\text{-Abstraction}$$

$$\frac{\Gamma \vdash M\colon\gamma \qquad x\#\Gamma}{\Gamma \vdash \lambda(x\colon\gamma').M\colon\gamma' \multimap \gamma}\ K\text{-Abstraction}$$

$$\frac{\Gamma \vdash M\colon(\gamma_1 + \gamma_2 + \cdots + \gamma_j) \multimap \gamma \qquad \forall k \in 1,2,\ldots,j.\Delta_k \vdash N\colon\gamma_k}{\Gamma + \sum_{k=1}^{j} \Delta_k \vdash MN\colon\gamma}\text{ Application}$$

Figure 6.4.1: Kfoury's type system $\boldsymbol{\lambda}$

taken into account) because it is trying to do three jobs at once (handling contractions, handling multiple different types for N, and the application itself). The contexts are also defined as sets (Kfoury actually defined them as partial functions defined over finite subsets of variable names, but the statement here in terms of sets with $x\#\Gamma$ side conditions is equivalent), meaning that no Exchange rule is needed. There is a good reason to use this sort of formulation; with so many rules combined, every production makes an irreversible syntactic change to the term, meaning that there is only the one derivation of any given term with any given type, greatly simplifying several proofs (e.g. coherence becomes degenerate). However, our aims are different from Kfoury's, so we will use a more traditional presentation with more but simpler productions.

The other major change we need to make is because we are studying ICA, rather than untyped lambda calculus. This means that we require an extra restriction; Kfoury's theory was designed to type terms like $(\lambda(x:((\mathsf{val} \multimap \mathsf{val}) \multimap (\mathsf{val} \multimap \mathsf{val})) + (\mathsf{val} \multimap \mathsf{val})).xx)(\lambda(y:\mathsf{val}).y):$ $(\mathsf{val} \multimap \mathsf{val})$, because such terms are strongly β-normalizing in untyped lambda calculus, but we wish to reject such terms, because they do not type in ICA. We do this with a new side condition $\theta \sim \theta'$, used in any rule that can generate a type of shape $\theta + \theta'$, that requires θ and θ' to have the same ICA type. We also need to add new features to the type system, because ICA has more features than untyped lambda calculus; we want tensors (because ICA has products), more than one base type, and constants. The resulting type system, Intersecting ICA, is shown in Figure 6.4.2.

It can be constructive to compare this type system to several of the other type systems we are working with. The comparison with Kfoury's $\boldsymbol{\lambda}$ is perhaps the most obvious; we can see that some major changes have been made, but the type system still works in fundamentally the same way. The split of K-Abstraction into Abstraction and Weakening is noticeable, but only a minor change (although one interesting change is that Kfoury disallowed Weakening in types of the form $\theta + \theta'$, whereas this type system allows it; Kfoury's motivation seems to have been studying the implications this had on weak β-normalization, something that does not concern us). The major difference, apart from the addition of constants and tensors, is that Application

$$\theta ::= \mathsf{exp}_J \mid \theta \multimap \theta \mid \theta \otimes \theta \mid \theta + \theta, \text{ where } J \subset \mathbb{N} \text{ is a finite set}$$
$$\Gamma ::= \text{sequence of } x\!:\!\theta$$
$$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$

$$\frac{x \neq y}{x\!:\!\theta \# y\!:\!\theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma :: \Gamma'\#\Delta} \qquad \frac{x\!:\!\theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{\mathsf{exp}_J \sim \mathsf{exp}_J} \qquad \frac{\theta_1 \sim \theta_2 \qquad \theta_1' \sim \theta_2'}{\theta_1 \otimes \theta_1' \sim \theta_2 \otimes \theta_2'} \qquad \frac{\theta_1 \sim \theta_2 \qquad \theta_1' \sim \theta_2'}{\theta_1' \multimap \theta_1 \sim \theta_2' \multimap \theta_2}$$

$$\frac{\theta_1 \sim \theta_2 \qquad \theta_1' \sim \theta_2}{\theta_1 + \theta_1' \sim \theta_2} \qquad \frac{\theta_1 \sim \theta_2 \qquad \theta_1 \sim \theta_2'}{\theta_1 \sim \theta_2 + \theta_2'}$$

$$\frac{}{x\!:\!\theta \vdash x\!:\!\theta}\ \text{Identity}$$

$$\frac{M\!:\!\theta \text{ is a constant}}{\vdash M\!:\!\theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M\!:\!\theta \qquad x\#\Gamma}{\Gamma :: x\!:\!\theta' \vdash M\!:\!\theta}\ \text{Weakening}$$

$$\frac{\Gamma :: x\!:\!\theta' :: y\!:\!\theta'' :: \Delta \vdash M\!:\!\theta}{\Gamma :: y\!:\!\theta'' :: x\!:\!\theta' :: \Delta \vdash M\!:\!\theta}\ \text{Exchange}$$

$$\frac{\Gamma :: x\!:\!\theta' :: \Delta \vdash M\!:\!\theta}{\Gamma :: \Delta \vdash \lambda(x\!:\!\theta').M\!:\!(\theta' \multimap \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M\!:\!(\theta' \multimap \theta) \qquad \Delta \vdash N\!:\!\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash MN\!:\!\theta}\ \text{Application}$$

$$\frac{\Gamma \vdash M\!:\!\theta \qquad \Delta \vdash N\!:\!\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash M \otimes N\!:\!(\theta \otimes \theta')}\ \text{Tensor}$$

$$\frac{\Gamma :: x\!:\!\theta' :: y\!:\!\theta'' \vdash M\!:\!\theta \qquad \theta' \sim \theta''}{\Gamma :: x\!:\!(\theta' + \theta'') \vdash M[x/y]\!:\!\theta}\ \text{Contraction}$$

$$\frac{x_1 : \theta_1 :: x_2 : \theta_2 :: \cdots x_j : \theta_j \vdash M : \theta_0 \qquad x_1 : \theta_1' :: x_2 : \theta_2' :: \cdots x_j : \theta_j' \vdash M : \theta_0' \qquad \forall k.\theta_k \sim \theta_k'}{x_1 : (\theta_1 + \theta_1') :: x_2 : (\theta_2 + \theta_2') :: \cdots :: x_j : (\theta_j + \theta_j') \vdash M : (\theta_0 + \theta_0')}\ \text{Intersection}$$

Figure 6.4.2: Intersecting ICA

$$\begin{array}{rcl}
\mathsf{skip} & : & \mathsf{com} \\
j_J & : & \mathsf{exp}_J \\
\mathsf{seq}_J & : & \mathsf{com} \multimap \mathsf{exp}_J \multimap \mathsf{exp}_J \\
\mathsf{par} & : & \mathsf{com} \multimap \mathsf{com} \multimap \mathsf{com} \\
\mathsf{if} & : & \mathsf{exp}_{\{0,1\}} \multimap \mathsf{exp}_J \multimap \mathsf{exp}_J \multimap \mathsf{exp}_J \\
\mathsf{uncurry}_{j,\theta'_k,\theta''_k,\theta} & : & (\sum_{k=1}^{j} \theta'_k \multimap (\sum_{k=1}^{j} \theta''_k \multimap \theta)) \multimap (\sum_{k=1}^{j} (\theta'_k \otimes \theta''_k) \multimap \theta) \\
\mathsf{op}_{J,\bullet} & : & \mathsf{exp}_J \multimap \mathsf{exp}_J \multimap \mathsf{exp}_J \\
\mathsf{newvar}_{j,J} & : & (\sum_{k=1}^{j} (\mathsf{exp}_J \otimes (\mathsf{exp}_J \multimap \mathsf{com})) \multimap \mathsf{com}) \multimap \mathsf{com}
\end{array}$$

Figure 6.4.3: Constants of Intersecting ICA

has been split into three rules: Intersection, that handles the type of N; Application; and Contraction, which handles combining variables in the contexts. Thus, this can be thought of as an "explicit contractions and intersections" version of Kfoury's type system. Also noteworthy is that we allow types of the form $\theta + \theta'$ for terms, not just free variables; this is necessary for the Intersection rule to follow the grammar for judgements.

Intersecting ICA can also be compared with Bounded ICA. It is not directly a generalization. The most obvious correspondence is to define $1 \cdot \theta$ as θ and $j \cdot \theta$ (for $j \geq 1$) as $(j-1) \cdot \theta + \theta$; we then discover that all but one of the rules of Bounded ICA become obviously admissible in Intersecting ICA, with most of them becoming the same in the two type systems, and Application of Bounded ICA becoming $j-1$ uses of Intersection followed by one use of Application in Intersecting ICA. However, there are two differences: types of the form $0 \cdot \theta$ in Bounded ICA do not map into Intersecting ICA (a difference that is mostly irrelevant, and trivial to work around by adding a "zero type" to Intersecting ICA such that $0 + \theta = \theta$ and $\forall M. M : 0$); and Subtyping from Bounded ICA has no obvious Intersecting ICA analogue. This latter difference is because both subtyping and intersection aim to work around the same problem; having no Subtyping rule in Intersecting ICA lets us compare the effectiveness of these two approaches.

Comparing Intersecting ICA to Bounded ICA is useful in another way: it gives us types for the constants. These types are shown in Figure 6.4.3. The main interest here is in the type of uncurry, which can distribute $+$ over $\otimes$, in much the same way that the uncurry of Bounded ICA can split a bound j from one tensor onto two separate arguments to a function.

Another comparison is that of Intersecting ICA with Affine ICA. This is perhaps the most striking comparison: all the rules of Affine ICA exist in Intersecting ICA, which differs only in the addition of Intersection and Contraction rules and types of the form $\theta + \theta'$. This tells us that Intersecting ICA is a straightforward generalization of Affine ICA, which is desirable but not particularly surprising. More usefully, though, it gives us a starting point for producing a semantics of Intersecting ICA; the subset of Intersecting ICA that corresponds to Affine ICA already has a semantics, and thus we merely need to define semantics for Intersection, Contraction, and $\theta + \theta'$. One remaining difference is in the type of newvar, but there is an obvious correspondence between the two types which will be formally explored later, and thus this turns out not to make much of a difference.

Finally, we can compare Intersecting ICA to ICA-without-semaphores-or-fix. The most interesting open question here is as to whether the two type systems admit the same terms; it seems likely (and Intersecting ICA was designed in order to aim for this goal), but has not been proved.

6.5 Eliminating bounded contraction

The previous sections of this chapter have been discussing type systems that describe terms that use bounded contraction. In this section, we wish to consider an application of these type systems: transforming terms to eliminate the use of this sort of contraction. The intended scenario is one in which we have an term in a language that allows bounded contraction (e.g. Bounded ICA), and wish to compile it into some other language with more restrictive contraction rules (e.g. Affine ICA), perhaps to allow the use of a simpler semantics, to gain a new representation of a form that might be easier to study (Kfoury's motivation in doing this was to gain a better understanding of β-reduction), or to enable the use of a construction (such as the construction by Ghica and myself in [19] for compiling SCI terms into hardware) that cannot cope with general contraction. This can be thought of as a general technique for *implementing* bounded contraction; given an implementation of a type system that does not support bounded contraction, the techniques in this section enable the construction of an implementation of the same type system but with bounded contraction added.

The way in which we implement bounded contraction is to copy a term; given a judgement like $x\!:\!3\cdot\mathsf{com}\vdash M\!:\!\theta$ or $x\!:\!(\mathsf{com}+\mathsf{com}+\mathsf{com})\vdash M\!:\!\theta$, the usual interpretation of the context, as contraction, is "*x* is used at most three times in *M*". However, an equally valid interpretation is "*M* requires three copies of *x*", and this interpretation does not imply the use of contraction anywhere; instead, we could consistently rewrite the rest of the derivation so that any abstraction on *x* becomes three abstractions, any applications of the resulting term become three applications with three different arguments, and so on. The result is a derivation of a different term, but one that is semantically equivalent (ideally, it would be possible to show a bijection between reductions in the operational semantics of the two terms). The name of this process depends on the resulting type system. In Kfoury's work, all forms of contraction are eliminated altogether, and thus the process is called *linearization* (although perhaps "affinization" would be a better name because the resulting terms still use Weakening, and thus are affine not linear). In the work I did on the subject with Dan Ghica,[20] we were looking to eliminate only bounded contraction, while preserving sequential contraction (which is discussed in Chapter 8); thus, the name "linearization" was inappropriate, and we called the process *serialization* instead (because the only remaining uses of contraction were in sequential contexts).

In this section, we consider two specific uses of this process: linearization of Bounded ICA into Affine ICA, and linearization of Intersecting ICA into Affine ICA. Other combinations of type systems are definitely useful, however; for example, in [20], Ghica and I serialized SCC into a very slight generalization of SCI (the generalization was necessary because the constants were different). There are likely numerous other combinations of type systems for which the process makes sense.

First, we consider linearization of Intersecting ICA, because this is one of the simpler cases

$$\theta ::= \mathsf{exp}_J \mid \theta \multimap \theta \mid \theta \otimes \theta \mid \theta + \theta, \text{ where } J \subset \mathbb{N} \text{ is a finite set}$$

$$\Gamma ::= \text{sequence of } x : \theta$$

$$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$

$$\overline{\mathsf{exp}_J} \triangleq \mathsf{exp}_J;\ \overline{\theta' \multimap \theta} \triangleq \overline{\theta'} \multimap \overline{\theta};\ \overline{\theta \otimes \theta'} \triangleq \theta \otimes \theta';\ \overline{\theta + \theta'} \triangleq \overline{\theta} \otimes \overline{\theta'}$$

$$\delta_{x:\theta',y:\theta'',\theta}(M') \triangleq \mathsf{uncurry}_{\overline{\theta'},\overline{\theta''},\overline{\theta}}(\lambda(x:\overline{\theta'}).\lambda(y:\overline{\theta''}).M')$$

$$\frac{x \neq y}{x:\theta \# y:\theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma :: \Gamma'\#\Delta} \qquad \frac{x:\theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{x:\theta \vdash x:\theta}\ \text{Identity}$$

$$\frac{M:\theta \text{ is a constant of Intersecting ICA, other than } \mathsf{newvar} \text{ or } \mathsf{uncurry}}{\vdash M \triangleright M : \theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M \triangleright M' : \theta \qquad x\#\Gamma}{\Gamma :: x:\theta' \vdash M \triangleright M' : \theta}\ \text{Weakening}$$

$$\frac{\Gamma :: x:\theta' :: y:\theta'' :: \Delta \vdash M \triangleright M' : \theta}{\Gamma :: y:\theta'' :: x:\theta' :: \Delta \vdash M \triangleright M' : \theta}\ \text{Exchange}$$

$$\frac{\Gamma :: x:\theta' :: \Delta \vdash M \triangleright M' : \theta}{\Gamma :: \Delta \vdash \lambda(x:\theta').M \triangleright \lambda(x:\overline{\theta'}).M' : (\theta' \multimap \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M \triangleright M' : (\theta' \multimap \theta) \qquad \Delta \vdash N \triangleright N' : \theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash MN \triangleright M'N' : \theta}\ \text{Application}$$

$$\frac{\Gamma \vdash M \triangleright M' : \theta \qquad \Delta \vdash N \triangleright N' : \theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash M \otimes N \triangleright M' \otimes N' : (\theta \otimes \theta')}\ \text{Tensor}$$

$$\frac{\Gamma :: x:\theta' :: y:\theta'' \vdash M \triangleright M' : \theta}{\Gamma :: x:(\theta' + \theta'') \vdash M[x/y] \triangleright (\delta_{x:\theta',y:\theta'',\theta}(M'))(x) : \theta)}\ \text{Contraction}$$

$$\frac{x_1:\theta_1 :: x_2:\theta_2 :: \cdots x_j:\theta_j \vdash M \triangleright M' : \theta_0 \qquad x_1:\theta_1' :: x_2:\theta_2' :: \cdots x_j:\theta_j' \vdash M \triangleright M'' : \theta_0'}{\begin{array}{c} x_1:(\theta_1 + \theta_1') :: x_1:(\theta_2 + \theta_2') :: \cdots :: x_j:(\theta_j + \theta_j') \vdash M \triangleright \\ \delta_{x_1:\theta_1,y_1:\theta_1',\theta_0}(\ldots \delta_{x_j:\theta_j,y_j:\theta_j',\theta}(M' \otimes M'') \ldots)(x_1) \ldots (x_j) : (\theta_0 + \theta_0') \end{array}}\ \text{Intersection}$$

Figure 6.5.1: An intermediate type system between Intersecting ICA and Affine ICA

$$\theta_1 \triangleq (\overline{\theta'} \multimap (\overline{\theta''} \multimap \overline{\theta})) = \overline{(\theta' \multimap (\theta'' \multimap \theta))}$$
$$\theta_2 \triangleq (\overline{\theta'} \otimes \overline{\theta''}) = \overline{(\theta' + \theta'')}$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{uncurry}_{\overline{\theta'},\overline{\theta''},\overline{\theta}} : (\theta_1 \multimap (\theta_2 \multimap \overline{\theta}))} \qquad \dfrac{\dfrac{\overline{\Gamma} :: x : \overline{\theta'} :: y : \overline{\theta''} \vdash M : \overline{\theta}}{\overline{\Gamma} :: x : \overline{\theta'} \vdash \lambda(y : \overline{\theta''}).M : (\overline{\theta''} \multimap \overline{\theta})}}{\overline{\Gamma} \vdash \lambda(x : \overline{\theta'}).\lambda(y : \overline{\theta''}).M : \theta_1}}{\overline{\Gamma} \vdash \mathsf{uncurry}_{\overline{\theta'},\overline{\theta''},\overline{\theta}}(\lambda(x : \overline{\theta'}).\lambda(y : \overline{\theta''}).M) : (\theta_2 \multimap \overline{\theta})} \qquad \dfrac{}{x : \theta_2 \vdash x : \theta_2}}{\overline{\Gamma} :: x : \theta_2 \vdash (\delta_{x:\theta',y:\theta'',\theta}(M))(x) : \overline{\theta}}$$

Figure 6.5.2: Proof of Lemma 6.5.1

of linearization: after all, seven of the rules of Intersecting ICA and Affine ICA are identical, thus there are only two rules we need to handle, Intersection and Contraction. The basic idea is very simple, that of consistently replacing $+$ with $\otimes$ and adapting the terms to match. In order to formalize the process, we use an intermediate type system, shown in Figure 6.5.1. As has occasionally been seen previously, this type system uses four parts to its judgement rather than three; however, in this type system, each judgement $\Gamma \vdash M \triangleright M' : \theta$ contains two terms, with M an Intersecting ICA term, and M' an Affine ICA term. We use an overline to represent the replacement of $+$ with $\otimes$, and extend it to contexts in the obvious manner; thus, the idea of this type system is that if $\Gamma \vdash M \triangleright M' : \theta$ is derivable in the intermediate type system, then $\Gamma \vdash M : \theta$ is derivable in Intersecting ICA and $\overline{\Gamma} \vdash M' : \overline{\theta}$ is derivable in Affine ICA, with M and M' being semantically equivalent. In most derivation rules, the same or equivalent transformations are applied to M and M'; however, both Contraction and Intersection work in terms of a subsidiary definition δ, whose purpose is to implement the contraction of free variables.

Lemma 6.5.1. *If $\overline{\Gamma :: x : \theta' :: y : \theta''} \vdash M : \overline{\theta}$ is derivable in Affine ICA, so is $\overline{\Gamma :: x : (\theta' + \theta'')} \vdash (\delta_{x:\theta',y:\theta'',\theta}(M))(x) : \overline{\theta}$.*

Proof. We give an explicit derivation, shown in Figure 6.5.2. □

Theorem 6.5.2. *If $\Gamma \vdash M : \theta$ is derivable in Intersecting ICA and does not use the* newvar *or* uncurry *constants, there is some M' such that $\Gamma \vdash M \triangleright M' : \theta$ is derivable in the intermediate type system; and if $\Gamma \vdash M \triangleright M' : \theta$ is derivable in the intermediate type system, $\Gamma \vdash M : \theta$ is derivable* *in Intersecting ICA.*

Proof. Each of the rules of the intermediate type system becomes a rule of Intersecting ICA when the M' part of each $\Gamma \vdash M \triangleright M' : \theta$ judgement is deleted; and for each rule of the intermediate system, given the rule, the premises, and the Γ, M, and θ of the conclusion, if $\Gamma \vdash M : \theta$ is the conclusion of the same rule in Intersecting ICA (with the M' deleted), there is at least one (in fact, exactly one) M' that can be used to make $\Gamma \vdash M \triangleright M' : \theta$ the conclusion of that rule in the intermediate type system. From there, the proof is a straightforward structural induction. □

Theorem 6.5.3. *If $\Gamma \vdash M \triangleright M' : \theta$ is derivable in the intermediate type system, then $\overline{\Gamma} \vdash M' : \overline{\theta}$ is derivable in Affine ICA.*

Proof. Proof is by structural induction. We find that the Identity, Constant, Weakening, Exchange, Abstraction, Application, and Tensor cases are all trivial. The case of Contraction is covered by Lemma 6.5.1. For Intersection, we first apply the Tensor rule of Affine ICA to combine the two premises, then use the result of Lemma 6.5.1 j times, together with uses of the Exchange rule to adjust the free variable lists into appropriate orders, to produce the desired conclusion. □

This approach works fine for everything but the higher-order constants, newvar and $\mathsf{uncurry}$. The issue with these constants is that their types in the two type systems differ, albeit in a relatively innocuous way. For example, we have $\mathsf{newvar}_{2,\{0\}} : ((((\mathsf{com} \otimes (\mathsf{com} \multimap \mathsf{com})) + (\mathsf{com} \otimes (\mathsf{com} \multimap \mathsf{com}))) \multimap \mathsf{com}) \multimap \mathsf{com})$ in Intersecting ICA, with the corresponding Affine ICA type being $(((\mathsf{com} \otimes (\mathsf{com} \multimap \mathsf{com})) \otimes (\mathsf{com} \otimes (\mathsf{com} \multimap \mathsf{com}))) \multimap \mathsf{com}) \multimap \mathsf{com}$; but the closest Affine ICA equivalent of the constant is $\mathsf{newvar}_{2,2,\{0\}} : ((((\mathsf{com} \otimes \mathsf{com}) \otimes ((\mathsf{com} \multimap \mathsf{com}) \otimes (\mathsf{com} \multimap \mathsf{com}))) \multimap \mathsf{com}) \multimap \mathsf{com})$, which is not the same type. Likewise, we have $\mathsf{uncurry}_{2,\theta'_k,\theta''_k,\theta} : ((((\theta'_1 + \theta'_2) \multimap ((\theta''_1 + \theta''_2) \multimap \theta))) \multimap (((\theta'_1 \otimes \theta''_1) + (\theta'_2 \otimes \theta''_2)) \multimap \theta))$ in Intersecting ICA; the corresponding Affine ICA type is $(((\overline{\theta'_1} \otimes \overline{\theta'_2}) \multimap ((\overline{\theta''_1} \otimes \overline{\theta''_2}) \multimap \overline{\theta}))) \multimap (((\overline{\theta'_1} \otimes \overline{\theta''_1}) \otimes (\overline{\theta'_2} \otimes \overline{\theta''_2})) \multimap \overline{\theta})$, which is not the type of any $\mathsf{uncurry}$ instance (the closest you can get is $(((\overline{\theta'_1} \otimes \overline{\theta'_2}) \multimap ((\overline{\theta''_1} \otimes \overline{\theta''_2}) \multimap \overline{\theta}))) \multimap (((\overline{\theta'_1} \otimes \overline{\theta'_2}) \otimes (\overline{\theta''_1} \otimes \overline{\theta''_2})) \multimap \overline{\theta})$).

We can note that the difference between the types, in each case, is simply that tensors appear in a different order or with different associativity. Thus, all we need to handle these constants is a general method, in Affine ICA, of re-associating and reversing the order of tensors within an expression. Although there is more than one way to do this, the categorical coherence results that are normally used to show coherence of multiple derivations for the same term can also be used to show equality of multiple terms of the same type:

Theorem 6.5.4. *Given any semantics of Affine ICA that is an instance of the categorical semantics in Figure 5.2.3, and for which* $[\![\mathsf{uncurry}]\!]$ *is formed entirely out of* α, α^{-1}, γ, ρ, ρ^{-1}, **abs**, **eval**, **id**, $\Rightarrow$, $\otimes$, *and morphism composition, then two closed Affine ICA terms are semantically equal (in that semantics) if both have the same type, that type is entirely parameterized on the right hand side of every* $\multimap$, *and both terms can be derived using no uses of the Weakening rule and no constants but* uncurry.

Proof. By structural induction on the derivations, both terms have denotations formed entirely out of α, α^{-1}, γ, ρ, ρ^{-1}, **abs**, **eval**, **id**, $\Rightarrow$, $\otimes$, and morphism composition; and both terms' denotations are transformations between the same types, and entirely parameterized on the right hand side of every $\Rightarrow$. Thus, by using an existing result from Kelly and MacLane [27, Theorem 2.4 on page 107], the two denotations must be equal. □

This result means that all methods of writing an "Intersecting ICA uncurry" in Affine ICA are effectively equal, and thus there is no need to pick one in particular. The result does not generalize as obviously to newvar, which is not entirely parameterized (it requires exp and com in specific places, and that each exp_J used among the ports has the same J); however, treating newvar's type as being made of opaque "read ports" and "write ports" indicates that there is effectively only one way to write newvar, too, because all read ports function in the same way, as do all write ports.

Thus, to linearize Intersecting ICA into Affine ICA, we have two choices. We could transform into Affine ICA exactly, by choosing families of Affine ICA terms that have the same types as Intersecting ICA's newvar and uncurry. Alternatively, we could just linearize into an

Affine-ICA-like type system that had Intersecting ICA's constants. In either case, the result is the same in terms of semantics; so if the reason we are linearizing is for simpler semantics, we can just use the second approach and avoid the need to worry unduly about the types of the constants. In practical uses of linearization and serialization, this tends to be enough; for example, the serialization in [20] by Ghica and myself serializes SCC into bSCI except with SCC's newvar; the second approach was actually necessary in this case, because SCC's newvar is not derivable in bSCI as the standard bSCI newvar is insufficiently powerful to emulate it (although this thesis suggests, in Section 8.1, an alternative newvar for bSCI which can).

The presence of tensors in Intersecting ICA makes the above linearization quite easy. We can use a similar approach for Bounded ICA; we can almost simply just transform a Bounded ICA derivation into a corresponding Intersecting ICA derivation, then use our existing results. However, Intersecting ICA is missing the Subtyping rule from Bounded ICA, and thus to linearize Bounded ICA into Affine ICA, we need an implementation of the Subtyping rule. Our algorithm is to "desubtype" Bounded ICA into Bounded ICA without subtyping, convert to Intersecting ICA, and then linearize from there.

For desubtyping, then, we need some way to start with a term $\Gamma \vdash M : \theta$ in Bounded ICA, and for any type θ' such that $\theta \leq \theta'$, find a term M' that is in some sense equivalent to M such that $\Gamma \vdash M : \theta'$ is derivable. Our algorithm for this is a defined recursively on the structure of $\leq$, and shown in Figure 6.5.3, which gives a definition of a Bounded ICA term $\mathbf{desub}_{\theta,\theta'} : 1 \cdot \theta \multimap \theta'$ for any $\theta \leq \theta'$, such that $\mathbf{desub}_{\theta,\theta'}(M)$ is in some sense equivalent to M. (The diagram also contains a proof that all such **desub** terms type in Bounded ICA, with the desired type; the type of uncurry is omitted to save space.) This equivalence is hard to formalize – the denotations of the two terms are *not* equal, because they have different types – but we can note that (by structural induction) it holds in any sense of equivalence for which the following two equivalences hold: $\lambda x.Mx \equiv M$ (a well-known rule of lambda calculus), and $\mathsf{uncurry}(\lambda x.\lambda y.x \otimes y)(z) \equiv z$ (which is very much how the uncurry constant is expected to operate, and which is implied by Theorem 6.5.4). In a call-by-name semantics, $\lambda x.Mx \equiv M$ is exactly an equivalence, so this

$$
\begin{array}{rcl}
\mathbf{desub}_{\mathsf{exp}_J,\mathsf{exp}_J} & \triangleq & \lambda(x\colon\mathsf{exp}_J).x \\
\mathbf{desub}_{\theta_1\otimes\theta_1',\theta_2\otimes\theta_2'} & \triangleq & \mathsf{uncurry}_{1,\theta_1,\theta_1',\theta_2\otimes\theta_2'}(\lambda(x\colon 1\cdot\theta_1).\lambda(y\colon 1\cdot\theta_1').(\mathbf{desub}_{\theta_1,\theta_2}(x)\otimes\mathbf{desub}_{\theta_1',\theta_2'}(y)) \\
\mathbf{desub}_{j_1\cdot\theta_1'\multimap\theta_1,j_2\cdot\theta_2'\multimap\theta_2} & \triangleq & \lambda(x\colon j_1\cdot\theta_1'\multimap\theta_1).\lambda(y\colon j_2\cdot\theta_2').\mathbf{desub}_{\theta_1,\theta_2}(x(\mathbf{desub}_{\theta_2',\theta_1'}(y)))
\end{array}
$$

$$
\dfrac{\dfrac{}{x\colon 1\cdot\mathsf{exp}_J\vdash x\colon\mathsf{exp}_J}}{\vdash\lambda(x\colon 1\cdot\mathsf{exp}_J).x\colon 1\cdot\mathsf{exp}_J\multimap\mathsf{exp}_J}
$$

$$
\dfrac{\vdash\mathsf{uncurry}_{1,\theta_1,\theta_1',\theta_2\otimes\theta_2'}\qquad
\dfrac{\dfrac{\dfrac{\dfrac{\vdash\mathbf{desub}_{\theta_1,\theta_2}\colon(1\cdot\theta_1\multimap\theta_2)\qquad\dfrac{}{x\colon 1\cdot\theta_1\vdash x\colon\theta_1}}{x\colon 1\cdot\theta_1\vdash\mathbf{desub}_{\theta_1,\theta_2}(x)\colon\theta_2}\qquad\dfrac{\vdash\mathbf{desub}_{\theta_1',\theta_2'}\colon(1\cdot\theta_1'\multimap\theta_2')\qquad\dfrac{}{y\colon 1\cdot\theta_1'\vdash y\colon\theta_1'}}{y\colon 1\cdot\theta_1'\vdash\mathbf{desub}_{\theta_1',\theta_2'}(y)\colon\theta_2'}}{x\colon 1\cdot\theta_1::y\colon 1\cdot\theta_1'\vdash\mathbf{desub}_{\theta_1,\theta_2}(x)\otimes\mathbf{desub}_{\theta_1',\theta_2'}(y)\colon(\theta_2\otimes\theta_2')}}{x\colon 1\cdot\theta_1\vdash\lambda(y\colon 1\cdot\theta_1').\mathbf{desub}_{\theta_1,\theta_2}(x)\otimes\mathbf{desub}_{\theta_1',\theta_2'}(y)\colon(1\cdot\theta_1'\multimap(\theta_2\otimes\theta_2'))}}{\vdash\lambda(x\colon 1\cdot\theta_1).\lambda(y\colon 1\cdot\theta_1').\mathbf{desub}_{\theta_1,\theta_2}(x)\otimes\mathbf{desub}_{\theta_1',\theta_2'}(y)\colon(1\cdot\theta_1\multimap(1\cdot\theta_1'\multimap(\theta_2\otimes\theta_2')))}}
{\vdash\mathsf{uncurry}_{1,\theta_1,\theta_1',\theta_2\otimes\theta_2'}(\lambda(x\colon 1\cdot\theta_1).\lambda(y\colon 1\cdot\theta_1').(\mathbf{desub}_{\theta_1,\theta_2}(x)\otimes\mathbf{desub}_{\theta_1',\theta_2'}(y))\colon((\theta_1\otimes\theta_1')\multimap(\theta_2\otimes\theta_2'))}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash\mathbf{desub}_{\theta_1,\theta_2}\colon(1\cdot\theta_1\multimap\theta_2)\qquad\dfrac{\dfrac{}{x\colon 1\cdot(j_1\cdot\theta_1'\multimap\theta_1)\vdash x\colon(j_1\cdot\theta_1'\multimap\theta_1)}\qquad\dfrac{\vdash\mathbf{desub}_{\theta_2',\theta_1'}\colon(1\cdot\theta_2'\multimap\theta_1')\qquad\dfrac{}{y\colon 1\cdot\theta_2'\vdash y\colon\theta_2'}}{y\colon 1\cdot\theta_2'\vdash\mathbf{desub}_{\theta_2',\theta_1'}(y)\colon\theta_1'}}{x\colon 1\cdot(j_1\cdot\theta_1'\multimap\theta_1)::y\colon j_1\cdot\theta_2'\vdash x(\mathbf{desub}_{\theta_2',\theta_1'}(y))\colon\theta_1}}{x\colon 1\cdot(j_1\cdot\theta_1'\multimap\theta_1)::y\colon j_1\cdot\theta_2'\vdash\mathbf{desub}_{\theta_1,\theta_2}(x(\mathbf{desub}_{\theta_2',\theta_1'}(y)))\colon\theta_2}}{x\colon 1\cdot(j_1\cdot\theta_1'\multimap\theta_1)::y\colon j_1\cdot\theta_2'::z\colon(j_2-j_1)\cdot\theta_2'\vdash\mathbf{desub}_{\theta_1,\theta_2}(x(\mathbf{desub}_{\theta_2',\theta_1'}(y)))\colon\theta_2}}{x\colon 1\cdot(j_1\cdot\theta_1'\multimap\theta_1)::y\colon j_2\cdot\theta_2'\vdash\mathbf{desub}_{\theta_1,\theta_2}(x(\mathbf{desub}_{\theta_2',\theta_1'}(y)))\colon\theta_2}}{x\colon 1\cdot(j_1\cdot\theta_1'\multimap\theta_1)\vdash\lambda(y\colon j_2\cdot\theta_2').\mathbf{desub}_{\theta_1,\theta_2}(x(\mathbf{desub}_{\theta_2',\theta_1'}(y)))\colon(j_2\cdot\theta_2'\multimap\theta_2)}}{\vdash\lambda(x\colon 1\cdot(j_1\cdot\theta_1'\multimap\theta_1)).\lambda(y\colon j_2\cdot\theta_2').\mathbf{desub}_{\theta_1,\theta_2}(x(\mathbf{desub}_{\theta_2',\theta_1'}(y)))\colon(1\cdot(j_1\cdot\theta_1'\multimap\theta_1)\multimap(j_2\cdot\theta_2'\multimap\theta_2))}
$$

Figure 6.5.3: Desubtyping in Bounded ICA

desubtyping algorithm works just fine for type systems like Bounded ICA that are designed for use with a call-by-name semantics.

In a sense, the fact that $\mathbf{desub}_{\theta,\theta'}(M)$ is derivable in Bounded ICA whenever M is means that the Subtyping rule of Bounded ICA is unnecessary. If each term M of function type were always written as $\lambda x.Mx$, and each term M of tensor type were always preceded by $\mathsf{uncurry}(\lambda x.\lambda y.x \otimes y)$, then Contraction and Weakening would be enough to gain the same functionality. The version of the type system that uses Subtyping is somewhat more readable, though, and thus easier to program in. As a side note, even in the absence of subtyping, the $\mathbf{desub}_{\theta,\theta}(M)$ transformation seems like a potentially useful one for gaining power in type systems; this is a direction that may be interesting to explore more thoroughly.

There are more complex cases of serialization in the literature. The serialization of SCC into bSCI (with SCC's constants) in [20] is mostly due to Ghica (I provided the algorithm for handling Subtyping, along similar lines to the algorithm above, but that was my only contribution); and it cannot be performed along similar lines as serializations that aim for Affine ICA. The fundamental issue is that bSCI has no tensors, and (in particular) no $\mathsf{uncurry}$ constant, meaning that δ cannot be expressed. Ghica's solution was to use a more complex transformation on contexts; instead of the $\overline{x : 2 \cdot (2 \cdot \mathsf{com} \multimap \mathsf{com})} \triangleq x : (((\mathsf{com} \otimes \mathsf{com}) \multimap \mathsf{com}) \otimes ((\mathsf{com} \otimes \mathsf{com}) \multimap \mathsf{com})$ transformation that is used when targeting Affine ICA, Ghica defined $\overline{x : 2 \cdot (2 \cdot \mathsf{com} \multimap \mathsf{com})} \triangleq x_1 : (\mathsf{com} \multimap (\mathsf{com} \multimap \mathsf{com})) :: x_2 : (\mathsf{com} \multimap (\mathsf{com} \multimap \mathsf{com}))$, an entirely different method of implementing the transformation. In Section 8.2, we find that there are other reasons to want tensors in bSCI; thus, it is worth considering just adding tensors to the language, and thus avoiding the need for this tensor-free algorithm. Still, it is worth bearing in mind that linearization and serialization are valuable techniques in a range of type systems, even ones where they might at first seem not to be viable. And this is important: it means that bounded contraction is a feature that can safely be added to almost any type system without fundamentally changing its nature, providing a way to make our type systems more powerful effectively for free.

Chapter 7

SYNCHRONIZED CONTRACTION

The use of bounded contraction, as seen in Chapter 6, enables code reuse (a feature missing from Affine ICA) without losing the ability to implement the program in a finite-state way. However, because this is implemented via copying terms, the code reuse only exists at the syntax level, and so is only really a convenience to the programmer; the copies will still exist in an implementation of the program, and so there is no runtime gain compared to simply writing the code out multiple times.

For practical use, we would therefore want a less wasteful form of contraction. We can observe that the method Affine ICA uses to ensure that only a finite amount of state is used is to associate all state with a term, and ensure each term is only evaluated once. However, this is a stronger restriction than is actually required; state is a renewable resource, in that once a term has finished executing, its state can be reused. Thus, we can make our programs less wasteful of resources via using a time-sharing policy for state, allowing multiple uses of a terms to use the same state so long as their uses do not overlap in time.

There are two main ways to capture the timing properties of a program. In Chapter 8, we will consider the "asynchronous" case in which time is driven by the execution of terms, i.e. nothing external determines the length of time a term takes to execute. However, we first consider the simpler "synchronous" case, where the time for which a term is allowed to execute is imposed on the term from outside.

7.1 Semiring-bounded Linear Logic

There are multiple ways in which we could introduce the notion of time into a type system. However, we want to aim for the most general method in this section, so as to avoid having to work out the theory separately for each case. As such, we consider the abstract notion of "time intervals", without (yet) placing a meaning on what those intervals represent. To be able to define contraction on base types, the relevant operations on these intervals are checking whether two intervals τ, τ' are *disjoint* (written $\tau \# \tau'$), and determining the *union* (written $\tau \oplus \tau'$, because we will later formalize time intervals as semirings) of two disjoint time intervals. The basic idea is that we can write "$x : \tau \cdot \theta$" for "the free variable $x : \theta$ is required during time interval τ"; and if we have $x : \tau \cdot \theta$ and $y : \tau' \cdot \theta$ with $\tau \# \tau'$, we can contract the two variables into a single variable $x : (\tau \oplus \tau') \cdot \theta$.

Higher-order terms are not quite so straightforward. As an example, suppose we we have two free variables $f : (\tau_1 \cdot (\tau_1' \cdot \theta' \multimap \theta))$ and $g : (\tau_2 \cdot (\tau_2' \cdot \theta' \multimap \theta))$; then (by the above definition) these terms can be contracted only if $\tau_1 \# \tau_2$ and $\tau_1' = \tau_2'$. In most call-by-name semantics, an argument to a function can evaluate only while the function itself is evaluating; so if τ_1' and τ_2' are taken to be time intervals in their own right, it is hard to see how a sensible definition of "disjoint" could allow two variables with non-base types to ever be contracted. The solution is to assume that τ_1' and τ_2' somehow define time intervals *relative to* τ_1 and τ_2; thus, they can be equal and yet describe different intervals. This means that we need a third operation on intervals, $\tau \odot \tau'$, which describes in an absolute sense the same interval that is described by τ' relative to τ.

We can now construct our type system, shown in Figure 7.1.1; it is parameterized by a semiring K which has identities $\mathbf{0}$, $\mathbf{1}$ and operations $\oplus$, $\odot$.[1] This type system happens to be more generally applicable than just to synchronization (thus the generic name); I discovered it upon noticing a general pattern among various other type systems, and a generalization of it was independently discovered by Brunel et al ([4, page 6]). The type system as given here differs

[1] I have previously presented this type system (unnamed) in [21].

$$\theta ::= \mathsf{exp}_J \mid \tau \cdot \theta \multimap \theta \mid \theta \otimes \theta, \text{ where } J \subset \mathbb{N} \text{ is a finite set, } \tau \in K \text{ is a time interval}$$
$$\Gamma ::= \text{sequence of } x\colon \tau \cdot \theta, \text{ where } \tau \in K \text{ is a time interval}$$
$$\tau \odot \varepsilon \triangleq \varepsilon;\ \tau \odot (x\colon \tau' \cdot \theta :: \Gamma) \triangleq x\colon (\tau \odot \tau') \cdot \theta :: (\tau \cdot \Gamma)$$
$$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$

$$\frac{x \neq y}{x\colon \theta \# y\colon \theta'} \qquad \frac{\Gamma \# \Delta \qquad \Gamma' \# \Delta}{\Gamma :: \Gamma' \# \Delta} \qquad \frac{x\colon \theta \# \Gamma}{x \# \Gamma}$$

$$\frac{}{x\colon \mathbf{1} \cdot \theta \vdash x\colon \theta}\ \text{Identity}$$

$$\frac{M\colon \theta \text{ is a constant}}{\vdash M\colon \theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M\colon \theta \qquad x \# \Gamma}{\Gamma :: x\colon \tau \cdot \theta' \vdash M\colon \theta}\ \text{Weakening}$$

$$\frac{\Gamma :: x\colon \tau' \cdot \theta' :: y\colon \tau'' \cdot \theta'' :: \Delta \vdash M\colon \theta}{\Gamma :: y\colon \tau'' \cdot \theta'' :: x\colon \tau' \cdot \theta' :: \Delta \vdash M\colon \theta}\ \text{Exchange}$$

$$\frac{\Gamma :: x\colon \tau \cdot \theta' :: \Delta \vdash M\colon \theta}{\Gamma :: \Delta \vdash \lambda(x\colon \tau \cdot \theta').M\colon (\tau \cdot \theta' \multimap \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M\colon (\tau \cdot \theta' \multimap \theta) \qquad \Delta \vdash N\colon \theta' \qquad \Gamma \# \Delta}{\Gamma :: (\tau \odot \Delta) \vdash MN\colon \theta}\ \text{Application}$$

$$\frac{\Gamma \vdash M\colon \theta \qquad \Delta \vdash N\colon \theta' \qquad \Gamma \# \Delta}{\Gamma :: \Delta \vdash M \otimes N\colon (\theta \otimes \theta')}\ \text{Tensor}$$

$$\frac{\Gamma :: x\colon \tau \cdot \theta' :: y\colon \tau' \cdot \theta' \vdash M\colon \theta}{\Gamma :: x\colon (\tau \oplus \tau') \cdot \theta' \vdash M[x/y]\colon \theta}\ \text{Contraction}$$

Figure 7.1.1: Semiring-bounded Linear Logic (call-by-name fragment)

from that in [4] in that it has no Subtyping rule, and restricts semiring elements to appearing on free variables and the left hand side of $\multimap$ connectives. Neither type system is named in either paper that introduced it, but the full type system has since (via informal agreement) become known as “Semiring-bounded Linear Logic” (by analogy with Bounded Linear Logic), and this fragment is the call-by-name fragment (“call-by-name SBLL”). Ghica gave a categorical semantics and coherence proof for this type system in [21]. (Interestingly, this proof does not require that K is left-distributive (i.e. it works even omitting the axiom $(\tau \oplus \tau') \odot \tau'' = (\tau \odot \tau'') \oplus (\tau' \odot \tau'')$); thus, if necessary, this type system can be trivially generalized via dropping that semiring axiom.)

However, there are problems with trying to use the type system for any of the features discussed in this thesis but synchronization. The language suffers from the polymorphism problem explained in Theorem 6.3.2; no matter what the choice of K, either (for any given ICA type) we can find some set of time intervals that are appropriate for any term with that ICA type, or else there are going to be some terms from typed lambda calculus that fail to type. In many cases, this causes practical and philosophical issues; for example, Bounded ICA is a subset of an instance of full SBLL (which contains a subtyping rule), and it fails to type some terms that use only bounded contraction. For synchronized contraction, the polymorphism problem is less of a problem because we would not *expect* every ICA term to type. A simple example is $\lambda f.\lambda x.f(f(x))$; in the usual case when the time interval for a function’s argument is not disjoint with that for a function itself, there is no way to contract the two copies of f. For synchronized contraction, however, this is what we want, because the two copies of f are clearly running concurrently and thus we cannot use the same state for both. It is possible to imagine a (presumably call-by-value) semantics in which a function’s argument finishes executing before the function itself starts, and in such a case, synchronized contraction becomes possible again.

Based on the considerations discussed above, we would expect to have a side condition $\tau \# \tau'$ on the Contraction rule. However, this causes multiple problems. One is categorical: a category cannot have a functor that only applies to certain objects or morphisms, and thus $\oplus$

$$\dfrac{\vdash M\colon((\tau_1 \oplus \tau_2)\odot\theta)\multimap\theta \qquad \dfrac{\vdash N\colon((\tau_3\oplus\tau_4)\odot\theta)\multimap\theta \qquad \vdash \kappa\colon\theta}{\vdash N(\kappa)\colon\theta}}{\vdash M(N(\kappa))\colon\theta}$$

Figure 7.1.2: Reused state in call-by-name SBLL

has to be defined everywhere, including a definition for $\tau \oplus \tau'$ when $\tau \# \tau'$. (The result will be an uninhabited interval that cannot be the type of any term.) The other is that, in many cases, such a side condition would not be sufficient to ensure that none of the state in an implementation of a term is used for multiple purposes at the same time. For a general example of the problem, see the partial derivation in Figure 7.1.2; here, we assume that some closed terms $M\colon((\tau_1 \oplus \tau_2)\odot\theta)\multimap\theta$ and $N\colon((\tau_3\oplus\tau_4)\odot\theta)\multimap\theta$ exist, as does some constant $\kappa\colon\theta$. A side-condition on the Contraction rule would, at best, require $\tau_1 \# \tau_2$ and $\tau_3 \# \tau_4$. However, if there is any state in the constant κ, it will be used at times $(\tau_1 \odot \tau_3)$, $(\tau_2 \odot \tau_3)$, $(\tau_1 \odot \tau_4)$, $(\tau_2 \odot \tau_4)$; and there are many reasonable choices for K for which (say) $\tau_1 \# \tau_2$ and $\tau_3 \# \tau_4$ do not together imply $(\tau_1 \odot \tau_4) \# (\tau_2 \odot \tau_3)$. Thus, although some sort of restriction is necessary in SBLL-based type systems, a $\tau \# \tau'$ side condition on Contraction often does not go far enough. We will consider some possible solutions throughout the following sections.

7.2 Hard real-time computation

The simplest example of a synchronous system is one in which time is measured in discrete, equally sized units; time is a completely absolute concept, and is not affected by anything that happens during program execution. Although it would be possible to use seconds, nanoseconds, etc., the theory of synchronous systems is entirely parametric on the actual unit of time used, so it is usual to introduce an abstract "clock tick" as the unit of time measurement; the idea is that there is some global timekeeper counting units of time, which program execution keeps to religiously. This view of the world is the *hard real-time* view; everything that happens takes some fixed length of time that is known in advance.

With this view, a time interval is defined by the moments in time at which it starts and ends; because everything is on a single global clock, there is only one timeline, so these values are just numbers. We need to carefully determine what those numbers mean, however, because of the need for relative time intervals in the type system. One method that works is for the a time interval to be represented as a fraction of the time taken up by the interval it is measured relative to, e.g. one possible time interval would mean "from halfway to three quarters of the way through the parent interval".[2] This is probably the only representation that works if the duration is considered to be part of the time interval, due to the need for semirings to have a unit. An alternative approach is for a time interval to be represented merely by its start time, and to have the duration instead be a feature of a base type; the advantage of this approach is that it fits more naturally onto the sort of constants that would normally be desired in such languages (the practical timing properties of many standard constants look very awkward in the previous notation, especially higher-order constants like newvar, because the (presumably) fixed length of time the implementation of the variable itself takes to run is not a constant fraction of the (potentially) varying length of time the code using the variable will take to run). We consider both approaches in this section: those where a time interval represents an affine transformation on another interval, and those where it is just a relative time. In the first approach, an interval υ is a pair $\langle t,d\rangle$, where t is the start time (as a real number from 0 to 1 inclusive), and d is the duration (from 0 to $1-t$ inclusive). In the latter approach, an interval υ is just a number; we will use nonnegative integers because those fit best with how practical hard real-time systems operate, but there is no mathematical reason why rational or real numbers could not be used. To be able to define $\odot$, both these structures need an associative operation; for integers, this is just addition, and for pairs:

Definition 7.2.1. Given two intervals $\langle t,d\rangle$, $\langle t',d'\rangle$, we define the *composition* of these intervals $\langle t,d\rangle\langle t',d'\rangle$ as $\langle t+dt',dd'\rangle$.

[2] I previously discussed this view of time intervals in [21, section 3], which uses an alternative notation. Some of the results in this section were also proved in that paper (albeit with different proofs); however, that paper focuses on a different aspect of real-time computation to this thesis.

Theorem 7.2.2. *Interval composition is associative, and has the identity* $\langle 0,1\rangle$.

Proof. For the identity, we have $\langle t,d\rangle\langle 0,1\rangle = \langle t+0d, 1d\rangle = \langle t,d\rangle = \langle 0+1t, 1d\rangle = \langle 0,1\rangle\langle t,d\rangle$. For associativity, we have:

$$\begin{aligned}
(\langle t_1,d_1\rangle\langle t_2,d_2\rangle)\langle t_3,d_3\rangle &= \langle t_1+d_1t_2, d_1d_2\rangle\langle t_3,d_3\rangle \\
&= \langle t_1+d_1t_2+d_1d_2t_3, d_1d_2d_3\rangle \\
&= \langle t_1+d_1(t_2+d_2t_3), d_1d_2d_3\rangle \\
&= \langle t_1,d_1\rangle\langle t_2+d_2t_3, d_2d_3\rangle \\
&= \langle t_1,d_1\rangle(\langle t_2,d_2\rangle\langle t_3,d_3\rangle)
\end{aligned}$$

□

In order for contraction to be meaningful, we cannot just use these intervals directly; we need some way to express, for instance, the type of "commands that take one tick, and start at cycle 0, 3, or 6". This implies that our semiring elements τ should be sets of intervals υ. It is actually more convenient to use multisets rather than sets; because $\tau \oplus \tau'$ needs to be defined even when $\tau \# \tau'$ fails to hold, it is much simpler to have $\tau \oplus \tau' \triangleq \tau \uplus \tau'$ (so that we can ensure that $\tau \oplus \tau$ (for nonempty τ) is uninhabited via observing repeated elements inside the multiset), than it would be to have $\tau \oplus \tau' = \tau \cup \tau'$, in which case $\tau \oplus \tau$ must be inhabited whenever τ is. We note that both definitions of an interval above (both pairs, and integers) form semigroups (that is, they have an associative operation which has an identity), and that a multiset of intervals is a function from intervals to integers, and thus from a semigroup to a semiring. Given any semigroup and semiring, there is a standard semiring structure (the "semigroup semiring") on functions from the semigroup to the semiring:

Definition 7.2.3. We define a semiring structure on the set K of functions τ from elements υ of a specified semigroup (with identity $\mathbf{e}$) to elements j of a specified semiring J, as follows:

- $\mathbf{0}_K(\upsilon) = \mathbf{0}_J$;
- $\mathbf{1}_K(\mathbf{e}) = \mathbf{1}_J$; $\mathbf{1}_K(\upsilon) = \mathbf{0}_J$ when $\upsilon \neq \mathbf{e}$;
- $(\tau \oplus_K \tau')(\upsilon) = \tau(\upsilon) \oplus_J \tau'(\upsilon)$;

For clarity, and because it is irrelevant, subscripts that represent the range of an exp have been omitted.

We define abbreviations $\mathbf{1} \triangleq [\langle 0,1\rangle]$, $\tau_1 \triangleq [\langle 0,0.5\rangle]$, $\tau_2 \triangleq [\langle 0.5,0.5\rangle]$, $\theta = \tau_1 \cdot \mathsf{exp} \multimap (\tau_2 \cdot \mathsf{exp} \multimap \mathsf{exp})$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f:\mathbf{1}\cdot\theta \vdash f:\theta} \quad \dfrac{\dfrac{\dfrac{}{g:\mathbf{1}\cdot\theta \vdash g:\theta} \quad \dfrac{}{x:\mathbf{1}\cdot\mathsf{exp} \vdash x:\mathsf{exp}}}{g:\mathbf{1}\cdot\theta :: x:\tau_1\cdot\mathsf{exp} \vdash g(x):(\tau_2\cdot\mathsf{exp}\multimap\mathsf{exp})} \quad \dfrac{}{\vdash 1:\mathsf{exp}}}{g:\mathbf{1}\cdot\theta :: x:\tau_1\cdot\mathsf{exp} \vdash g(x)(1):\mathsf{exp}}}{f:\mathbf{1}\cdot\theta :: g:\tau_1\cdot\theta :: x:[\langle 0,0.25\rangle]\cdot\mathsf{exp} \vdash f(g(x)(1)):(\tau_2\cdot\mathsf{exp}\multimap\mathsf{exp})} \quad \dfrac{\dfrac{\dfrac{}{h:\mathbf{1}\cdot\theta \vdash h:\theta} \quad \dfrac{}{\vdash 2:\mathsf{exp}}}{h:\mathbf{1}\cdot\theta \vdash h(2):(\tau_2\cdot\mathsf{exp}\multimap\mathsf{exp})} \quad \dfrac{}{y:\mathbf{1}\cdot\mathsf{exp} \vdash y:\mathsf{exp}}}{h:\mathbf{1}\cdot\theta :: y:\tau_2\cdot\mathsf{exp} \vdash h(2)(y):\mathsf{exp}}}{\dfrac{f:\mathbf{1}\cdot\theta :: g:\tau_1\cdot\theta :: h:\tau_2\cdot\theta :: x:[\langle 0,0.25\rangle]\cdot\mathsf{exp} :: y:[\langle 0.75,0.25\rangle]\cdot\mathsf{exp} \vdash f(g(x)(1))(h(2)(y)):\mathsf{exp}}{\dfrac{f:\mathbf{1}\cdot\theta :: g:(\tau_1 \uplus \tau_2)\cdot\theta :: x:[\langle 0,0.25\rangle]\cdot\mathsf{exp} :: y:[\langle 0.75,0.25\rangle]\cdot\mathsf{exp} \vdash f(g(x)(1))(g(2)(y)):\mathsf{exp}}{f:\mathbf{1}\cdot\theta :: g:(\tau_1 \uplus \tau_2)\cdot\theta :: x:[\langle 0,0.25\rangle,\langle 0.75,0.25\rangle]\cdot\mathsf{exp} \vdash f(g(x)(1))(g(2)(x)):\mathsf{exp}}}}$$

(a) Durations on intervals

We define abbreviations $\theta_1 = [0]\cdot\mathsf{exp}_1 \multimap ([1]\cdot\mathsf{exp}_1 \multimap \mathsf{exp}_2)$, $\theta_2 = [0]\cdot\mathsf{exp}_2 \multimap ([1]\cdot\mathsf{exp}_2 \multimap \mathsf{exp}_3)$.

$$\dfrac{\dfrac{\dfrac{}{f:[0]\cdot\theta_2 \vdash f:\theta_2} \quad \dfrac{\dfrac{\dfrac{}{g:[0]\cdot\theta_1 \vdash g:\theta_1} \quad \dfrac{}{x:[0]\cdot\mathsf{exp}_1 \vdash x:\mathsf{exp}_1}}{g:[0]\cdot\theta_1 :: x:[0]\cdot\mathsf{exp}_1 \vdash g(x):([1]\cdot\mathsf{exp}_1\multimap\mathsf{exp}_2)} \quad \dfrac{}{\vdash 1:\mathsf{exp}_1}}{g:[0]\cdot\theta_1 :: x:[0]\cdot\mathsf{exp}_1 \vdash g(x)(1):\mathsf{exp}_2}}{f:[0]\cdot\theta_2 :: g:[0]\cdot\theta_1 :: x:[0]\cdot\mathsf{exp}_1 \vdash f(g(x)(1)):([1]\cdot\mathsf{exp}_2\multimap\mathsf{exp}_3)} \quad \dfrac{\dfrac{\dfrac{}{h:[0]\cdot\theta_1 \vdash h:\theta_1} \quad \dfrac{}{\vdash 2:\mathsf{exp}_1}}{h:[0]\cdot\theta_1 \vdash h(2):([1]\cdot\mathsf{exp}_1\multimap\mathsf{exp}_2)} \quad \dfrac{}{y:[0]\cdot\mathsf{exp}_1 \vdash y:\mathsf{exp}_1}}{h:[0]\cdot\theta_1 :: y:[1]\cdot\mathsf{exp}_1 \vdash h(2)(y):\mathsf{exp}_2}}{\dfrac{f:[0]\cdot\theta_2 :: g:[0]\cdot\theta_1 :: h:[1]\cdot\theta_1 :: x:[0]\cdot\mathsf{exp}_1 :: y:[2]\cdot\mathsf{exp}_1 \vdash f(g(x)(1))(h(2)(y)):\mathsf{exp}_3}{f:[0]\cdot\theta_2 :: g:[0,1]\cdot\theta_1 :: x:[0,2]\cdot\mathsf{exp}_1 \vdash f(g(x)(1))(g(2)(x)):\mathsf{exp}_3}}$$

(b) Durations on base types

Figure 7.2.1: Derivations of $f(g(x)(1))(g(2)(x))$

- $(\tau \odot_K \tau')(\upsilon) = \bigoplus_{\upsilon'\upsilon''=\upsilon} \tau(\upsilon') \odot_J \tau'(\upsilon'')$.

In the specific case where $J = \mathbb{N}$ (and thus K is a multiset), we have $\mathbf{0} = []$, $\mathbf{1} = [\mathbf{e}]$, $\tau \oplus \tau' = \tau \uplus \tau'$ (as the above discussion suggests should be the case), and $\odot$ as the *convolution* operation between functions (which also comes up in many unrelated contexts, notably signal processing).

As an example of how these languages function, consider the term $f(g(x)(1))(g(2)(x))$. (This sort of term is common in practice, especially if f or g is some sort of arithmetic operation that is too complex to be a constant, such as division or floating point addition.) Derivations of this term in the two languages we are considering are shown in Figure 7.2.1.

These derivations make the difference between the two languages clear. When durations are part of the interval, we learn what timings each term must have as a proportion of the whole; for instance, x needs to be valid for the first and last quarter of the execution of the term, and evaluate 4 times as quickly as the term as a whole. This approach lets us know, given how fast the program as a whole needs to complete, how fast each individual sub-term needs to complete. However, in many cases we have the converse problem: we know how long each constant takes to complete, and want to know how long the program as a whole will take. In this situation, placing the durations on the base types makes more sense.

The derivations above have one other difference, unrelated to the difference between the type systems: in Figure 7.2.1a, I chose the durations to avoid overlapping (i.e. the term is entirely sequential), whereas Figure 7.2.1b has some parallelism: the two copies of g overlap in time. This is not disallowed by the type system; whether or not it makes sense for any particular variable depends on the definition of that variable. If g is a fast arithmetic operation, then in practice, the type $[0,1] \cdot \theta_1$ will typically be reasonable for it; a likely implementation of $g \triangleq \lambda x.\lambda y.g(x)(y)$ would be to start evaluating x immediately and y after one cycle, store the output of x for one cycle while evaluating y, then perform the arithmetic on x and y and return it. Although two executions of g as a whole overlap, no individual part of g is ever trying to do two things at once, as is shown in Table 7.1. This method of overlapping executions of a term via exploiting the fact that there is no overlap once the term is expanded is known as *pipelining*,

Time	Starting at time 0	Starting at time 1
0	Evaluate x	
1	Store x, evaluate y, return	Evaluate x
2		Store x, evaluate y, return

Table 7.1: Pipelining behaviour of arithmetic

and is a common design technique in digital hardware. One big advantage of synchronized contraction is that pipelining does not need to be represented as a special case; it is a natural property of the type system.

We also need to consider what sort of side-condition is needed on this type system in order to ensure that the synchronized contraction we use does not try to use state for two purposes at the same time. When durations belong to intervals, a definition of $\tau\#\tau'$ as "for all $\upsilon \in \tau$, $\upsilon' \in \tau'$, the intervals υ, υ' do not overlap", and a side-condition $\tau\#\tau'$ on Contraction, is sufficient:

Definition 7.2.4. We define $\langle t,d\rangle\#\langle t',d'\rangle$ iff $t+d \leq t' \vee t'+d' \leq t$; and $\tau\#\tau'$ iff $\forall\upsilon,\upsilon'.(\upsilon\#\upsilon') \vee \tau(\upsilon)\tau'(\upsilon') = 0$.

Theorem 7.2.5. *If $\tau_1\#\tau_2$ then $\forall\tau.(\tau_1 \odot \tau)\#\tau_2$.*

Proof. Assume for implication that $\tau_1\#\tau_2$. Assume for contradiction that we can find υ_1, υ_2 such that neither $(\tau_1 \odot \tau)(\upsilon_1)\tau_2(\upsilon_2) = 0$ nor $\upsilon_1\#\upsilon_2$. By the definition of $\odot$, there are some υ_3, υ_4 such that $\tau_1(\upsilon_3)\tau(\upsilon_4) \neq 0$ and $\upsilon_3\upsilon_4 = \upsilon_1$. Thus, $\tau_1(\upsilon_3) \neq 0$ and $\tau_2(\upsilon_2) \neq 0$, meaning that (because $\tau_1\#\tau_2$) $\upsilon_3\#\upsilon_2$. Let $\upsilon_j = \langle t_j, d_j\rangle$. We have two cases:

- $t_3+d_3 \leq t_2$: then $t_1+d_1 = t_3+d_3t_4+d_3d_4 = t_3+d_3(t_4+d_4) \leq t_3+d_3 \leq t_2$ (by definition, $t_4+d_4 \leq 1$ and $d_3 \geq 0$).
- $t_2+d_2 \leq t_3$: then $t_2+d_2 \leq t_3 \leq t_3+d_3t_4 = t_1$ (because $d_3 \geq 0$ and $t_4 \geq 0$).

Both of these cases give us $\upsilon_1\#\upsilon_2$, contradicting the assumption. Thus the assumption is false, meaning that $\forall\tau.(\tau_1 \odot \tau)\#\tau_2$. □

Corollary 7.2.6. *If $\tau_1\#\tau_2$ and $\tau_3\#\tau_4$ then $\tau_1 \odot \tau_3\#\tau_2 \odot \tau_3$ and $\tau_1 \odot \tau_3\#\tau_2 \odot \tau_4$.*

Proof. Applying the above result twice, we get $\tau_1 \odot \tau_3 \# \tau_2$ and then $\forall \tau. \tau_1 \odot \tau_3 \# \tau_2 \odot \tau$. $\square$

However, just because $\tau \# \tau'$ is sufficient to ensure that the executions of terms never overlap, this does not mean it is necessarily the right side-condition. One issue is that it is undesirably restrictive, in that it disallows any sort of pipelining. Another is that there is no obvious way to apply it to the other type system we are discussing here, where the durations are on base types.

An alternative is to change the type system a bit more radically. Even without any sort of side-condition on contraction, the type system already handles free variables correctly with respect to time intervals; the issue is with constants. Thus, we can introduce a new section of the judgement, that keeps track of how each constant is used. The resulting type system is shown in Figure 7.2.2, and is equivalent to call-by-name SBLL in which all constants are replaced by free variables, and those free variables are never contracted or abstracted. (This equivalence also directly produces a categorical semantics for constant-tracking call-by-name SBLL, which is coherent for the same reason that the categorical semantics of call-by-name SBLL is.) Because constants are now treated much the same way as free variables, our constants are now handled just as correctly as our free variables are. This form is particularly useful when durations are part of the base types, because then, if a constant is fully pipelinable, our required side condition becomes very simple, "no multiset in Ψ contains repeated elements" (i.e. $\nexists \tau \in \Psi . \exists \upsilon . \tau(\upsilon) \geq 2$).

One final observation is that nothing about the type system where durations are part of the base types requires times to be expressed as integers; it would be possible to pick any semigroup and use it for the times and durations. One particularly interesting family of choices is that of the integers modulo a constant. This has just as much power as using integers directly, but adds another possibility: the implementation of terms that have a duration that can vary at runtime (so long as that duration is fixed modulo a constant), such as while. This seems like it may eventually lead to a type system that allows the automatic inference of pipelining within arbitrary terms, which seems like a promising direction for future research.

$$\theta ::= \mathsf{exp}_J \mid \tau \cdot \theta \multimap \theta \mid \theta \otimes \theta, \text{ where } J \subset \mathbb{N} \text{ is a finite set, } \tau \in K \text{ is a time interval}$$
$$\Gamma ::= \text{sequence of } x\colon \tau \cdot \theta, \text{ where } \tau \in K \text{ is a time interval}$$
$$\Psi ::= \text{sequence of } \tau, \text{ where } \tau \in K \text{ is a time interval}$$
$$\tau \odot \varepsilon \triangleq \varepsilon;\ \tau \odot (x\colon \tau' \cdot \theta :: \Gamma) \triangleq x\colon (\tau \odot \tau') \cdot \theta :: (\tau \cdot \Gamma);\ \tau \odot (\tau' :: \Psi) = \tau \odot \tau' :: (\tau \odot \Psi)$$
$$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$

$$\frac{x \neq y}{x\colon \tau \cdot \theta \# y\colon \tau' \cdot \theta'} \qquad \frac{\Gamma \# \Delta \qquad \Gamma' \# \Delta}{\Gamma :: \Gamma' \# \Delta} \qquad \frac{x\colon \tau \cdot \theta \# \Gamma}{x \# \Gamma}$$

$$\frac{}{\varepsilon \mid x\colon \mathbf{1} \cdot \theta \vdash x\colon \theta}\ \text{Identity}$$

$$\frac{M\colon \theta \text{ is a constant}}{\mathbf{1} \mid \varepsilon \vdash M\colon \theta}\ \text{Constant}$$

$$\frac{\Psi \mid \Gamma \vdash M\colon \theta \qquad x \# \Gamma}{\Psi \mid \Gamma :: x\colon \tau \cdot \theta' \vdash M\colon \theta}\ \text{Weakening}$$

$$\frac{\Psi \mid \Gamma :: x\colon \tau' \cdot \theta' :: y\colon \tau'' \cdot \theta'' :: \Delta \vdash M\colon \theta}{\Psi \mid \Gamma :: y\colon \tau'' \cdot \theta'' :: x\colon \tau' \cdot \theta' :: \Delta \vdash M\colon \theta}\ \text{Exchange}$$

$$\frac{\Psi \mid \Gamma :: x\colon \tau \cdot \theta' :: \Delta \vdash M\colon \theta}{\Psi \mid \Gamma :: \Delta \vdash \lambda(x\colon \tau \cdot \theta').M\colon (\tau \cdot \theta' \multimap \theta)}\ \text{Abstraction}$$

$$\frac{\Psi \mid \Gamma \vdash M\colon (\tau \cdot \theta' \multimap \theta) \qquad \Psi' \mid \Delta \vdash N\colon \theta' \qquad \Gamma \# \Delta}{\Psi :: (\tau \odot \Psi') \mid \Gamma :: (\tau \odot \Delta) \vdash MN\colon \theta}\ \text{Application}$$

$$\frac{\Psi \mid \Gamma \vdash M\colon \theta \qquad \Psi' \mid \Delta \vdash N\colon \theta' \qquad \Gamma \# \Delta}{\Psi :: \Psi' \mid \Gamma :: \Delta \vdash M \otimes N\colon (\theta \otimes \theta')}\ \text{Tensor}$$

$$\frac{\Psi \mid \Gamma :: x\colon \tau \cdot \theta' :: y\colon \tau' \cdot \theta' \vdash M\colon \theta}{\Psi \mid \Gamma :: x\colon (\tau \oplus \tau') \cdot \theta' \vdash M[x/y]\colon \theta}\ \text{Contraction}$$

Figure 7.2.2: Constant-tracking call-by-name SBLL

7.3 Tags versus tensors

In the previous section, we looked at the use of call-by-name SBLL for tracking time intervals. The type system is similar to several other type systems we have considered in this thesis, and several pre-existing type systems, in that it seeks to place a "tag" (in this case, a time interval) on terms in order to record information about it. In this thesis generally, we have been using the notation $\tau \cdot \theta$ for tags (where necessary, changing the notation of type systems such as SCC to make it clear that they are of this form).

When a type system designed for use with call-by-name languages has no form of tensor or product, it typically uses a grammar in which tags only appear on the left hand side of a function arrow (this is based on linear logic, in which when modeling call-by-name, $\theta \to \theta$ is defined as $!\theta \multimap \theta$; modalities more precise than linear logic's !, such as tags, appear in the same position). For example, the grammar of call-by-name SBLL gives us function arrows of shape $(\tau \cdot \theta) \multimap \theta$; and the original paper on SCC gives us the grammar "β is a base type, $\theta ::= \beta \mid \gamma \to \theta$, $\gamma ::= \theta^n$",[18, page 9] which in the notation of this thesis is "σ is a base type, $\gamma ::= \sigma \mid \theta \to \gamma$, $\theta ::= j \cdot \gamma$" (note that the symbols θ and γ have swapped meanings). Interestingly, SCC did not contain a pair-like construct in the original paper on the subject. Later papers discussing SCC tended to add a product operation (to match that in bSCI); [19, page 3] by Ghica and myself added products into SCC by replacing "$\gamma ::= \sigma \mid \theta \to \gamma$" with "$\gamma ::= \sigma \mid \theta \to \gamma \mid \theta \times \theta$", but this appears to have been a typographical error; by our next paper ([20, page 4]) on the subject, we had corrected it to (in the notation of this thesis) "$\gamma ::= \sigma \mid \theta \to \gamma \mid \gamma \times \gamma$".

If a tag can appear only on the left-hand side of a function arrow, then, as the history of definitions of SCC perhaps unintentionally illustrates, there is no obvious way to handle the interaction of tensors and tags. The following argument explains the problem: using (as usual) θ for a tagged type and γ for an untagged type, we can infer that because (in such type systems) we have $\gamma ::= \theta \to \gamma$, i.e. $\gamma ::= \tau \cdot \gamma \to \gamma$, we must also have $\gamma ::= \tau \cdot \gamma \to (\tau \cdot \gamma \to \gamma)$. According to normal categorical rules, we therefore expect $\gamma ::= (\tau \cdot \gamma \otimes \tau \cdot \gamma) \to \gamma$ (because we expect currying/uncurrying to have no semantic effect), and therefore $\gamma ::= (\theta \otimes \theta) \to \gamma$ and thus $\theta ::=$

$\theta \otimes \theta$. Although this produces an entirely reasonable type system, it is one that automatically disallows a substantial subset of ICA terms (or even Affine ICA terms) from typing within the type system, because no types of shape $- \to (- \otimes -)$ exist no matter whether you try to replace the holes with θ or γ types; a tensor is a θ type, and the right hand side of a function arrow is a γ type.

This limitation has some practical implications; for example, the "obvious" way to implement an array in an ICA-like type system is to give the array a type along the lines of $\mathsf{exp}_{J'} \to (\mathsf{exp}_J \otimes (\mathsf{exp}_J \to \mathsf{com}))$, but this does not map onto any type in such a type system (the closest you can get is $(\mathsf{exp}_{J'} \to \mathsf{exp}_J) \otimes (\mathsf{exp}_{J'} \to (\mathsf{exp}_J \to \mathsf{com}))$ with appropriate tags added, which is a syntactically different type).

One way to avoid this situation would be to instead put tags on both sides of a function arrow: instead of types like $\tau \cdot (\tau' \cdot \gamma_1 \to \gamma_2) \to \gamma_3$, you would use a different grammar ("$\gamma ::= \sigma$, $\theta ::= \tau \cdot \gamma \mid \theta \to \theta \mid \theta \otimes \theta$") which would produce types like $((\tau \odot \tau') \cdot \gamma_1 \to \tau \cdot \gamma_2) \to \mathbf{1} \cdot \gamma_3$. I refer to this sort of type system as a "premultiplied" type system, because instead of inferring a judgement $\Gamma \vdash N : (\tau' \cdot \gamma_1 \to \gamma_2)$ and then applying it to a term $\Delta \vdash M : (\tau \cdot (\tau' \cdot \gamma_1 \to \gamma_2) \to \gamma_3)$ to get $\tau \odot \Gamma :: \Delta \vdash MN : \gamma_3$, *multiplying* the term by τ as part of the Application rule, you instead derive the judgement $\tau \odot \Gamma \vdash N : ((\tau \odot \tau') \cdot \gamma_1 \to \tau \cdot \gamma_2)$ directly, and use a simpler Application rule which does no arithmetic. This sort of type system is simpler in many ways, but runs into a major stumbling block: there is no obvious definition for contraction for higher-order terms.

The obvious method to implement contraction in a premultiplied type system is to add the types point-wise, i.e. allowing $(\tau_1'' \cdot \theta'' \multimap \tau_1' \cdot \theta') \multimap \tau_1 \cdot \theta$ to contract with $(\tau_2'' \cdot \theta'' \multimap \tau_2' \cdot \theta') \multimap \tau_2 \cdot \theta$ to produce $((\tau_1'' \oplus \tau_2'') \cdot \theta'' \multimap (\tau_1' \oplus \tau_2') \cdot \theta') \multimap (\tau_1 \oplus \tau_2) \cdot \theta$. However, this loses the link between a function and its argument; in a type of shape, e.g. , $(\upsilon_1 \oplus \upsilon_2 \oplus \upsilon_3 \oplus \upsilon_4) \cdot \theta' \multimap (\upsilon_5 \oplus \upsilon_6) \cdot \theta$, it is unclear which of υ_5, υ_6 correspond to which of υ_1, υ_2, υ_3, υ_4; and worse, it is not necessarily the case that one of the former corresponds to exactly two of the latter. The end result is that you can have a higher-order function that expects two functions, one that needs its argument in three different time intervals, and one that only needs its argument in one time interval, and use the same function (that uses its argument in two time intervals) for both arguments. It is thus hard to see how it would be possible to produce a reasonable semantics for this sort of contraction rule; nor am I aware of any other contraction rule that might potentially work better.

Given that non-premultiplication leads to problems with tensors, and premultiplication leads to problems with contraction, it is interesting to see how the type systems we have looked at so far deal with tensors and contraction:

- Affine ICA does not have any sort of intersection typing nor tagging, meaning that it is meaningless to ask whether the tags/intersections go inside or outside a tensor.

- Bounded ICA and SCC always place contraction bounds on an entire product or tensor (e.g. $2 \cdot (\mathsf{com} \otimes \mathsf{com}) \multimap \mathsf{com}$ in Bounded ICA, $2 \cdot (\mathsf{com} \times \mathsf{com}) \rightarrow \mathsf{com}$ in SCC). This means that there is no problem placing a product or tensor on the right hand side of an arrow. The left hand side is a little harder to deal with, but not much, because the tags can be considered to distribute over the tensor; for example, in Bounded ICA, $2 \cdot (\mathsf{com} \otimes \mathsf{com}) \multimap \mathsf{com}$ is isomorphic to $2 \cdot \mathsf{com} \multimap (2 \cdot \mathsf{com} \multimap \mathsf{com})$. The main drawback here is that we necessarily lose information when uncurrying a term like $2 \cdot \mathsf{com} \multimap (3 \cdot \mathsf{com} \multimap \mathsf{com})$; we must first use Subtyping to convert it to $3 \cdot \mathsf{com} \multimap (3 \cdot \mathsf{com} \multimap \mathsf{com})$, and then uncurry it to $3 \cdot (\mathsf{com} \otimes \mathsf{com}) \multimap \mathsf{com}$. In Section 6.3, we saw that the use of Subtyping to fix problems with the type system is limited and will fail on some higher-order terms. However, the type systems have this problem even in the absence of tensors, so what we have here is a solution to the tensor problem that at least does not introduce any extra problems that SCC does not have already.

- Along similar lines to SCC, the call-by-name fragment of SBLL requires both halves of a tensor to be tagged the same way, i.e. $\tau \cdot (\mathsf{com} \otimes \mathsf{com}) \multimap \mathsf{com}$. This is bad enough when the tags represent the quantity of a resource used (as with Bounded ICA). The use with time intervals is even worse, because in addition to the problems mentioned so far, we also find that standard constants like $\mathsf{seq}_{\tau_1,\tau_2,J} : \tau_1 \cdot \mathsf{com} \multimap (\tau_2 \cdot \mathsf{com} \multimap \mathsf{com})$ $(\tau_1 \# \tau_2)$ can never be uncurried, as the $\tau_1 \# \tau_2$ side condition implies $\tau_1 \neq \tau_2$ except in the trivial case where $\tau_1 = \tau_2 = \mathbf{0}$. SCC's workaround using Subtyping does not help here. This type system can thus be seen to have the reverse of the problems with bSCI that are explained in Section 8.1: in bSCI, there is no way to write seq in curried form, and when using call-by-name SBLL for synchronized contraction, there is no way to write it in uncurried form. This would cause problems later on when we consider sequential contraction, the sort of contraction bSCI uses; we will discover that it is synchronized contraction with a side condition, and thus having our type systems for synchronized contraction being necessarily syntactically incompatible with bSCI is obviously undesirable.

- Intersecting ICA does not use tags; however, intersection types have many of the same problems as tags, in that they overlap in a special case ($\theta + \theta$ behaves like $2 \cdot \theta$). It does not place restrictions on whether intersections go inside or outside tensors: $((\mathsf{com} + \mathsf{com}) \otimes (\mathsf{com} + \mathsf{com})) \multimap \mathsf{com}$ and $((\mathsf{com} \otimes \mathsf{com}) + (\mathsf{com} \otimes \mathsf{com})) \multimap \mathsf{com}$ are both legal types (syntactically different, but isomorphic). This gives a lot of flexibility in what types are available; types like $(\mathsf{com} + \mathsf{com}) \multimap (\mathsf{com} + \mathsf{com})$ are syntactically valid, and even inhabited (Figure 7.3.1 derives a term of this type), meaning that in Intersecting ICA, the left-hand side of a function arrow is not necessarily defined relative to the right-hand side. However, this flexibility does not extend to higher-order types; for example, $\lambda x.x$ cannot be typed with a type of shape $(\theta_1 \multimap \theta_2) \multimap ((\mathsf{com} \multimap \mathsf{com}) + (\mathsf{com} \multimap \mathsf{com}))$. Additionally, the type system is not really premultiplied; if you want to derive a type for MN and M has type $(\theta_1 + \theta_2) \to \theta$, you derive $\Gamma_1 \vdash N : \theta_1$ and $\Gamma_2 \vdash N : \theta_2$, then use Intersection then Application, which is superficially similar to premultiplication in that a term of type $\theta_1 + \theta_2$ is derived, but different in that terms of type θ_1 and θ_2 are also derived.

From the survey above, we can see that the call-by-name fragment of SBLL has the tensor problem (in addition to the polymorphism problem); and we can also see that Intersecting ICA provides a potential solution to both problems. We will consider the polymorphism problem

$$\dfrac{\dfrac{}{x\colon \mathsf{com} \vdash x\colon \mathsf{com}} \qquad \dfrac{}{x\colon \mathsf{com} \vdash x\colon \mathsf{com}}}{\dfrac{x\colon (\mathsf{com}+\mathsf{com}) \vdash x\colon (\mathsf{com}+\mathsf{com})}{\vdash \lambda x.x\colon ((\mathsf{com}+\mathsf{com}) \multimap (\mathsf{com}+\mathsf{com}))}}$$

Figure 7.3.1: $(\mathsf{com}+\mathsf{com}) \multimap (\mathsf{com}+\mathsf{com})$ is inhabited in Intersecting ICA

$$\gamma ::= \mathsf{exp}_J \mid \theta \multimap \theta \mid \theta \otimes \theta, \text{ where } J \subset \mathbb{N} \text{ is a finite set}$$
$$\theta ::= \tau \cdot \gamma, \text{ where } \tau \in K \text{ is a time interval}$$
$$\Gamma ::= \text{sequence of } x\colon \theta, \text{ where } \tau \in K \text{ is a time interval}$$
$$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$
$$\tau \cdot \varepsilon \triangleq \varepsilon;\ \tau \cdot (x\colon \tau' \cdot \gamma :: \Gamma) \triangleq x\colon (\tau \odot \tau') \cdot \gamma :: (\tau \cdot \Gamma)$$

$$\frac{x \neq y}{x\colon \theta \# y\colon \theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma :: \Gamma'\#\Delta} \qquad \frac{x\colon \theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{x\colon \theta \vdash x\colon \theta}\ \text{Identity}$$

$$\frac{M\colon \theta \text{ is a constant}}{\vdash M\colon \theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M\colon \theta \qquad x\#\Gamma}{\Gamma :: x\colon \theta' \vdash M\colon \theta}\ \text{Weakening}$$

$$\frac{\Gamma :: x\colon \theta' :: y\colon \theta'' :: \Delta \vdash M\colon \theta}{\Gamma :: y\colon \theta'' :: x\colon \theta' :: \Delta \vdash M\colon \theta}\ \text{Exchange}$$

$$\frac{\Gamma :: x\colon \theta' :: \Delta \vdash M\colon \theta}{\Gamma :: \Delta \vdash \lambda(x\colon \theta').M\colon \mathbf{1} \cdot (\theta' \multimap \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M\colon \tau \cdot \gamma}{\tau' \cdot \Gamma \vdash M\colon (\tau' \odot \tau) \cdot \theta}\ \text{Tagging}$$

$$\frac{\Gamma \vdash M\colon \mathbf{1} \cdot (\theta' \multimap \theta) \qquad \Delta \vdash N\colon \theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash MN\colon \theta}\ \text{Application}$$

$$\frac{\Gamma \vdash M\colon \theta \qquad \Delta \vdash N\colon \theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash M \otimes N\colon \mathbf{1} \cdot (\theta \otimes \theta')}\ \text{Tensor}$$

$$\frac{\Gamma :: x\colon \tau \cdot \gamma' :: y\colon \tau' \cdot \gamma' \vdash M\colon \theta}{\Gamma :: x\colon (\tau \oplus \tau') \cdot \gamma' \vdash M[x/y]\colon \theta}\ \text{Contraction}$$

Figure 7.3.2: SBLL (flexible fragment)

$$\dfrac{\Gamma \vdash M : \mathbf{1} \cdot (\tau \cdot \gamma' \multimap \theta) \qquad \dfrac{\Delta \vdash N : \mathbf{1} \cdot \gamma'}{\tau \cdot \Delta \vdash N : (\tau \odot \mathbf{1}) \cdot \gamma} \qquad \Gamma \# \Delta}{\Gamma :: \tau \cdot \Delta \vdash MN : \theta}$$

Figure 7.3.3: Call-by-name SBLL's Application rule is admissible in the flexible fragment

in the next section, because at least when applied to hard real-time systems, it is not actually a problem. Applying just the solution to the tensor problem from Intersecting ICA to the call-by-name fragment of SBLL we get the type system in Figure 7.3.2 (which can also be generalized to track constants if necessary). This is also a fragment of SBLL (the main difference is that it now contains the pr rule from [4, page 6] explicitly, rather than only allowing it to be used implicitly as part of an Application), but a larger one than call-by-name SBLL is:

Theorem 7.3.1. *All terms derivable in call-by-name SBLL (Figure 7.1.1) can also be derived in the type system in Figure 7.3.2 with the same choice of semiring K, and with the same types (except for the addition of* $\mathbf{1}\cdot$ *tags wherever they are syntactically required).*

Proof. Proof is by structural induction.

The base cases are Identity and Constant, both of which are the same in the two type systems (up to the addition of $\mathbf{1}\cdot$ tags). For the inductive cases, Weakening, Exchange, Abstraction, and Contraction are the same up to the addition of $\mathbf{1}\cdot$ tags. The only remaining case is the Application rule of call-by-name SBLL; Figure 7.3.3 proves this to be admissible in the type system of Figure 7.3.2 via direct construction. ($\Gamma\#\Delta$ is true iff $\Gamma\#\tau\cdot\Delta$ because # only looks at variable names; and $\tau \odot \mathbf{1} = \tau$ because K is a semiring.) □

I call this the "flexible fragment" of SBLL, because it is a superset of call-by-name SBLL, but allows the two halves of a tensor to be tagged differently. It thus seems likely to be an improved alternative to call-by-name SBLL in cases where the extra complexity in the type system does not cause a problem.

7.4 Local timescales

Instead of looking as time as absolute and global, an alternative view is to consider synchronization as something that happens locally. In this case, two terms can be on the same timescale as each other, but do not have to be. Time thus becomes a preorder; an event might or might not happen unambiguously before or after another. This view is not particularly useful in a hard real-time setting, in which you can determine which of two events happens first by counting cycles from the start of the program. However, it becomes much more useful in an environment where terms take an unknown length of time to execute, because it makes it possible to synchronize terms despite a lack of precise timing information.

A simple example is $w,x,y,z \vdash \mathsf{par}(\mathsf{par}(\mathsf{seq}(x)(y))(\mathsf{seq}(w)(z)))(\mathsf{seq}(y)(x))$ (which may be easier to read in Verity syntax, as `((x;y)||(w;z))||(y;x))`. This term can be expected to type in a synchronized setting, because we can choose types such that the first use of x and second use of y stop executing at the same time, with the first use of y and second use of x starting at that point. Meanwhile, w stops executing when z starts executing, but we don't have (or need) any information about the timing of that event, compared to the timing of the switchover between x and y.

This sort of synchronization gives a lot of flexibility, which needs to be reflected in the type system. In the previous section, we could make use of the restrictions provided by the use of an absolute timescale to use an appropriate instance of call-by-name or flexible SBLL. However, if we try doing the same with a local-timescale-based type system, we run into trouble. Consider the following term:

$$\lambda f.((\lambda g.\mathsf{seq}(f(\lambda x_1.\lambda y_1.\lambda z_1.g(x_1)(\mathsf{seq}(y_1)(z_1)))))(f(\lambda x_2.\lambda y_2.\lambda z_2.g(\mathsf{seq}(x_2)(y_2))(z_2))))(\mathsf{seq}))$$

We would expect to be able to type this term using synchronized contraction, because the only sharing involved is of f (whose arguments in the two uses are semantically identical, and thus should be able to be given identical types because any sensible semantics for seq

is associative), and of g (which is seq), with both shared terms being used in two entirely disjoint time periods (due to the outermost seq). However, we run into trouble trying to calculate types (no tensors are involved here, so we'll use the call-by-name fragment). The type of f must be of the form $(\tau_1 \oplus \tau_2) \cdot (\tau_3 \cdot (\tau_4 \cdot \mathsf{com} \multimap \tau_5 \cdot \mathsf{com} \multimap \tau_6 \cdot \mathsf{com} \multimap \mathsf{com}) \multimap \mathsf{com})$; thus, x_j has type $\tau_j \odot \tau_3 \odot \tau_4$, y_j has type $\tau_j \odot \tau_3 \odot \tau_5$, and z_j has type $\tau_j \odot \tau_3 \odot \tau_6$. Meanwhile, g must have a type of the form $(\tau_7 \oplus \tau_8) \cdot (\tau_9 \cdot \mathsf{com} \multimap \tau_{10} \cdot \mathsf{com} \multimap \mathsf{com})$, giving x_1 a type $\tau_1 \odot \tau_3 \odot \tau_7 \odot \tau_9$, $\mathsf{seq}(y_1)(z_1)$ a type $\tau_1 \odot \tau_3 \odot \tau_7 \odot \tau_{10}$, $\mathsf{seq}(x_2)(y_2)$ a type $\tau_2 \odot \tau_3 \odot \tau_8 \odot \tau_9$, and z_2 a type $\tau_2 \odot \tau_3 \odot \tau_8 \odot \tau_{10}$. This combination defeats most attempts at finding a reasonable-looking semiring. For example, we might consider it reasonable to assume that seq only has types of the form $\tau_{\mathsf{seq}} \cdot \mathsf{com} \multimap \tau'_{\mathsf{seq}} \cdot \mathsf{com} \multimap \mathsf{com}$ for which $\forall \tau . \tau \odot \tau_{\mathsf{seq}} \subset \tau$ and $\forall \tau . \tau \odot \tau'_{\mathsf{seq}} \subset \tau$ (using $\subset$ to indicate inclusion of time intervals), and that $\forall \tau, \tau', \tau''.(\tau \odot \tau' \subset \tau \odot \tau''$ iff $\tau' \subset \tau'')$ and $\forall \tau, \tau', \tau''.(\tau' \odot \tau \subset \tau'' \odot \tau$ iff $\tau' \subset \tau'')$; but this gives $\tau_7 \odot \tau_9 = \tau_4 \subset \tau_8 \odot \tau_9$ and $\tau_8 \odot \tau_{10} = \tau_6 \subset \tau_7 \odot \tau_{10}$, which together imply $\tau_7 \subset \tau_8$ and $\tau_8 \subset \tau_7$.

One possible approach to this problem is to look for a more complex semiring that violates the assumptions given in the last paragraph; what is basically necessary is a semiring for which τ can, in some sense, behave differently when calculating $\tau' \odot \tau$ and when calculating $\tau'' \odot \tau$. Such a semiring would have to be very general, perhaps something along the lines of "the set of functions that map sets of real numbers onto subsets of those sets" (although possibly not that specific example, because it doesn't obviously have a semiring structure). However, developing a semiring that general is difficult; partly because there is no obvious reason why the semiring would be distributive (even though we don't need left-distributivity for SBLL, we do need right-distributivity, i.e. $\tau \odot (\tau' \oplus \tau'') = (\tau \odot \tau') \oplus (\tau \odot \tau'')$), partly because it involves uncountable infinities, making definitions much harder, and partly because type inference is unlikely to be decidable for a type system that general. Thus, although it does not seem impossible that such an approach could succeed, it is dubious that the result would be practically useful.

An alternative approach is to replace SBLL with a more general type system that is a better fit for local synchronization. We can observe that the problems with the example term we have been discussing are effectively caused by the fact that in the type of g, τ_9 and τ_{10} had to have some *specific* value, rather than being allowed to vary between the two instances of g, and likewise, τ_4, τ_5, and τ_6 had to be the same for both uses of f. If g could be polymorphic, then it could adjust the time allocation for its first and second arguments so as to be able to fit τ_5 and τ_6 into its second argument on the first call, and τ_4 and τ_5 into its first argument on the second call. Likewise, if f could be polymorphic, it could expand τ_4 to fit the entire first argument of g on the first call, and shrink it so that both τ_4 and τ_5 could fit into the first argument of g on the second call. Thus, we've effectively just run into a variant of the polymorphism problem we've seen repeatedly in earlier chapters; because synchronized contraction is not meant to be able to type *all* terms, Theorem 6.3.2 does not apply directly, but even though SBLL thus has not been proved outright inapplicable, it certainly seems like there should be an easier way.

The solution we adopt is the same as the one we used for bounded contraction: we generalize the type system to use intersection typing. In the case of bounded contraction, a contraction bound in Bounded ICA was an integer, whereas Intersecting ICA did not use contraction bounds. One way to think about the difference between those two type systems is that types in Bounded ICA are of the form $(1+1+\cdots+1)\cdot\theta$, whereas types in Intersecting ICA are of the form $(1\cdot\theta_1)+(1\cdot\theta_2)+\cdots+(1\cdot\theta_j)$, with the "1·" being omitted as unnecessary because no bounds other than 1 exist. We can do a similar transformation to the flexible fragment of SBLL; instead of addition being an operation inside the semiring, it becomes an operation on types. Thus, instead of parameterizing our logic by a semiring, we no longer have any additive structure, leaving us with an identity $\mathbf{1}$, and an associative operation $\odot$ for which $\mathbf{1}$ is a left and right identity, i.e. a semigroup. We also still need the # relation between elements of the semigroup, that specifies when two semigroup elements are considered disjoint, in order to be able to put an appropriate side condition on contraction. The resulting family of type systems, Intersection/Semigroup-bounded Logic (or ISBLL), is shown in Figure 7.4.1. However, we still don't lose the finite-state nature of our type system: Intersecting ICA can be linearized into Affine ICA, and ISBLL is a subset of Intersecting ICA:

$$\gamma ::= \mathsf{exp}_J \mid \theta \multimap \theta \mid \theta + \theta \mid \theta \otimes \theta,\ \text{where } J \subset \mathbb{N} \text{ is a finite set}$$
$$\theta ::= \tau \cdot \gamma,\ \text{where } \tau \in K \text{ is a time interval}$$
$$\Gamma ::= \text{sequence of } x\colon\theta$$
$$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$
$$\tau \odot \varepsilon \triangleq \varepsilon \qquad \tau \odot (x\colon\tau'\cdot\gamma :: \Gamma) \triangleq x\colon((\tau\odot\tau')\cdot\gamma) :: (\tau\odot\Gamma)$$

$$\frac{x \neq y}{x\colon\theta \# y\colon\theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma :: \Gamma'\#\Delta} \qquad \frac{x\colon\theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{\tau\cdot\mathsf{exp}_J \sim \tau'\cdot\mathsf{exp}_J} \qquad \frac{\theta_1\sim\theta_2 \qquad \theta_1'\sim\theta_2'}{\tau\cdot(\theta_1\otimes\theta_2)\sim\tau'\cdot(\theta_1'\otimes\theta_2')} \qquad \frac{\theta_1\sim\theta_2 \qquad \theta_1'\sim\theta_2'}{\tau_1\cdot(\theta_1'\multimap\theta_1)\sim\tau_2\cdot(\theta_2'\multimap\theta_2)}$$

$$\frac{\theta_1\sim\theta_2 \qquad \theta_1'\sim\theta_2}{\tau\cdot(\theta_1+\theta_1')\sim\theta_2} \qquad \frac{\theta_1\sim\theta_2 \qquad \theta_1\sim\theta_2'}{\theta_1\sim\tau\cdot(\theta_2+\theta_2')}$$

$$\frac{\tau_1\#\tau_2}{\tau_1\cdot\gamma_1\#\tau_2\cdot\gamma_2} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1'\#\theta_2}{\mathbf{1}\cdot(\theta_1+\theta_1')\#\theta_2} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1\#\theta_2'}{\theta_1\#\mathbf{1}\cdot(\theta_2+\theta_2')}$$

$$\frac{\theta_1\#\theta_2 \qquad \theta_1'\#\theta_2}{\mathbf{1}\cdot(\theta_1\otimes\theta_1')\#\theta_2} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1\#\theta_2'}{\theta_1\#\mathbf{1}\cdot(\theta_2\otimes\theta_2')} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1'\#\theta_2}{\mathbf{1}\cdot(\theta_1'\multimap\theta_1)\#\theta_2} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1\#\theta_2'}{\theta_1\#\mathbf{1}\cdot(\theta_2'\multimap\theta_2)}$$

$$\frac{}{x\colon\theta\vdash x\colon\theta}\ \text{Identity} \qquad \frac{M\colon\theta \text{ is a constant}}{\vdash M\colon\theta}\ \text{Constant}$$

$$\frac{\Gamma\vdash M\colon\theta \qquad x\#\Gamma}{\Gamma :: x\colon\theta'\vdash M\colon\theta}\ \text{Weakening} \qquad \frac{\Gamma :: x\colon\theta' :: y\colon\theta'' :: \Delta\vdash M\colon\theta}{\Gamma :: y\colon\theta'' :: x\colon\theta' :: \Delta\vdash M\colon\gamma}\ \text{Exchange}$$

$$\frac{\Gamma :: x\colon\theta' :: \Delta\vdash M\colon\theta}{\Gamma :: \Delta\vdash\lambda(x\colon\theta').M\colon\mathbf{1}\cdot(\theta'\multimap\theta)}\ \text{Abstraction}$$

$$\frac{\Gamma\vdash M\colon\tau\cdot\gamma}{\tau'\odot\Gamma\vdash M\colon(\tau'\odot\tau)\cdot\gamma}\ \text{Tagging}$$

$$\frac{\Gamma\vdash M\colon\mathbf{1}\cdot(\theta'\multimap\theta) \qquad \Delta\vdash N\colon\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta\vdash MN\colon\theta}\ \text{Application}$$

$$\frac{\Gamma\vdash M\colon\theta \qquad \Delta\vdash N\colon\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta\vdash M\otimes N\colon\mathbf{1}\cdot(\theta\otimes\theta')}\ \text{Tensor}$$

$$\frac{\begin{array}{c} x_1:\theta_1 :: \ldots :: x_j:\theta_j\vdash M\colon\theta_0 \\ x_1:\theta_1' :: \ldots :: x_j:\theta_j'\vdash M\colon\theta_0' \end{array} \qquad \bigwedge_{k=0}^{j}\theta_k\sim\theta_k' \qquad \bigwedge_{k=0}^{j}\theta_k\#\theta_k'}{x_1:\mathbf{1}\cdot(\theta_1+\theta_1') :: x_2:\mathbf{1}\cdot(\theta_2+\theta_2') :: \ldots :: x_j:\mathbf{1}\cdot(\theta_j+\theta_j')\vdash M\colon\mathbf{1}\cdot(\theta_0+\theta_0')}\ \text{Intersection}$$

$$\frac{\Gamma :: x\colon\theta' :: y\colon\theta''\vdash M\colon\theta \qquad \theta'\sim\theta'' \qquad \theta'\#\theta''}{\Gamma :: x\colon\mathbf{1}\cdot(\theta'+\theta'')\vdash M[x/y]\colon\theta}\ \text{Contraction}$$

Figure 7.4.1: Intersection/Semigroup-bounded Linear Logic

Theorem 7.4.1. *All terms that are derivable in any ISBLL instance have an Intersecting ICA type.*

Proof. Intersecting ICA is an instance of ISBLL (specifically, one where K is the trivial semigroup with one element $\mathbf{1}$ for which $\mathbf{1} \odot \mathbf{1} = \mathbf{1}$ and $\mathbf{1}\#\mathbf{1}$); after substituting this definition of K into Figure 7.4.1, we get Figure 6.4.2, plus a no-op Tagging rule that derives a judgement from itself. If for any ISBLL derivation (in any semiring, not just this K), we consistently replace all tags with $\mathbf{1}$, the derivation remains valid except that side conditions of the form $\tau\#\tau'$ will fail to hold unless $\mathbf{1}\#\mathbf{1}$. The derivation is thus valid in any ISBLL instance for which $\mathbf{1}\#\mathbf{1}$, and thus in Intersecting ICA. $\square$

In order to produce an actual type system for locally-timed synchronized contraction, we need to make a concrete choice for the semigroup K. We do this by choosing an arbitrary countably infinite set $\mathbb{A}$; then, elements of the semigroup are sequences that alternate between elements of $\mathbb{A}$, and $^+$ or $^-$, starting with an element of $\mathbb{A}$. (For notational purposes, we will assume that $\mathbb{A}$ is the set $\{\mathsf{u}_1, \mathsf{u}_2, \mathsf{u}_3, \ldots\}$ in this thesis, but any countably infinite set would work.) For example, ε, $\mathsf{u}_1^+\mathsf{u}_2^-$, and $\mathsf{u}_1^+\mathsf{u}_1^+\mathsf{u}_1^+$ are all elements of our semigroup K of time intervals. For an intuition about the meaning of these intervals, consider ε to be the entire length of time the program runs, and for all $\tau \in K$, $\upsilon \in \mathbb{A}$, think of $\tau :: \upsilon$ as being an arbitrary instant of time within τ, and $\tau :: \upsilon^-$ as being the period of time from the start of τ to $\tau :: \upsilon$, with $\tau :: \upsilon^+$ being the period of time from $\tau :: \upsilon$ to the end of τ.

The actual semigroup and disjointness structure is as follows: $\mathbf{1} \triangleq \varepsilon$; $\tau \odot \tau' \triangleq \tau :: \tau'$; and $\forall \tau, \upsilon, \tau_1, \tau_2.(\tau :: \upsilon^+ :: \tau_1)\#(\tau :: \upsilon^- :: \tau_2)$ (with two time intervals being disjoint only if they can be collectively expressed in this form). This obviously obeys the semigroup axioms, and has intuitive behaviour for # (two time intervals are disjoint if there is a point in time for which one interval falls entirely before that point, and one interval falls entirely after that point).

We also need appropriate constants, shown in Figure 7.4.2 (the sum in the definition of newvar is constructed via adding $\theta_1, \theta_2, \ldots \theta_j$ as $\varepsilon \cdot (\theta_1 + \varepsilon \cdot (\theta_2 + \ldots \varepsilon \cdot (\theta_{j-1} + \theta_j) \ldots)))$. These mostly follow the familiar patterns we've seen with preceding type systems, but now have added

$$\begin{aligned}
\mathsf{par} &: \varepsilon\cdot(\varepsilon\cdot\mathsf{com}\to\varepsilon\cdot(\varepsilon\cdot\mathsf{com}\to\varepsilon\cdot\mathsf{com}))\\
\mathsf{seq}_{J,\upsilon} &: \varepsilon\cdot(\upsilon^-\cdot\mathsf{com}\to\varepsilon\cdot(\upsilon^+\cdot\mathsf{exp}_J\to\varepsilon\cdot\mathsf{exp}_J))\\
\mathsf{if}_{J,\upsilon} &: \varepsilon\cdot(\upsilon^-\cdot\mathsf{exp}_{\{0,1\}}\to\varepsilon\cdot(\upsilon^+\cdot\mathsf{exp}_J\to\varepsilon\cdot(\upsilon^+\cdot\mathsf{exp}_J\to\mathsf{exp}_J)))\\
\mathsf{uncurry}_{\theta',\theta'',\theta} &: \varepsilon\cdot(\varepsilon\cdot(\theta'\to\varepsilon\cdot(\theta''\to\theta))\to\varepsilon\cdot(\varepsilon\cdot(\theta'\otimes\theta'')\to\theta))\\
\mathsf{op}_{J,\bullet,\upsilon} &: \varepsilon\cdot(\upsilon^-\cdot\mathsf{exp}_J\to\varepsilon\cdot(\upsilon^+\cdot\mathsf{exp}_J\to\varepsilon\cdot\mathsf{exp}_J))\\
\mathsf{skip} &: \varepsilon\cdot\mathsf{com}\\
j_J &: \varepsilon\cdot\mathsf{exp}_J \qquad (j\in\mathbb{N}, j\in J)\\
\mathsf{while}_\upsilon &: \varepsilon\cdot(\upsilon^-\cdot\mathsf{exp}_{\{0,1\}}\to\varepsilon\cdot(\upsilon^+\cdot\mathsf{com}\to\varepsilon\cdot\mathsf{com}))\\
\mathsf{newvar}_{J,A} &: \varepsilon\cdot(\textstyle\sum_{\langle\tau,\tau'\rangle\in A}(\tau\cdot\mathsf{exp}_J\otimes\tau'\cdot(\varepsilon\cdot\mathsf{exp}_J\to\varepsilon\cdot\mathsf{com}))\to\varepsilon\cdot\mathsf{com})\to\varepsilon\cdot\mathsf{com}\\
&\quad (A\subset K\times K, A \text{ is finite})
\end{aligned}$$

Figure 7.4.2: Constants used for ISBLL

tags: in parallel contexts, we use tags of ε; and in sequential contexts, we parameterize the constant by an element υ of $\mathbb{A}$, with υ^- and υ^+ being used to tag the arguments that cannot happen simultaneously. We also now have a while constant. This is something of a special case. The type system typically handles terms like $\lambda x.\mathsf{seq}(x)(\mathsf{seq}(x)(x))$ that execute their arguments multiple times by giving them types like $(\mathsf{u}_1^-\cdot\mathsf{com}+\mathsf{u}_1^+\mathsf{u}_2^-\cdot\mathsf{com}+\mathsf{u}_1^+\mathsf{u}_2^+\cdot\mathsf{com})\to\mathsf{com}$, where the type of the argument is a large intersection. In order to be consistent with the types of other terms, a while loop would conceptually need to be an infinite contraction, which is not allowed in the grammar. Nonetheless, the type for while is safe, in that it does not allow unwanted contraction. There are two ways to see this. One is to note that if while were limited to any finite number of iterations, any term that typed with the above type for while would also type if while had a type that explicitly listed all the iterations. The other is a semantic argument: the semantics of while means that multiple iterations of the loop cannot overlap with each other, and thus can share safely regardless of what the type system has to say.

It is worth also considering what an implementation of ISBLL would look like. In particular, the main unusual features of ISBLL are the Intersection, Contraction and Tagging rules. We can think of a tag as representing a condition about the global state of a program, and thus a term

$M : \tau \cdot \gamma$ as representing "$M : \mathbf{1} \cdot \gamma$, but which only runs while τ is true". Tagging can thus be seen as a locking operation; we can consider there to be an initially empty global set of "running tags". $[\![\tau' \odot \Gamma \vdash M : (\tau' \odot \tau) \cdot \gamma]\!]$ is then, informally, "wait until τ' does not contradict any other running tag; add τ' to the set of running tags; do $[\![\Gamma \vdash M : \tau \cdot \gamma]\!]$; remove τ' from the set of running tags". (Two tags τ, τ' "contradict" if $\tau \# \tau'$.) This sort of locking algorithm normally needs to be careful to avoid deadlocks, but in the case of local timescales, they are impossible due to the hierarchical structure of the locks; a term $\Gamma \vdash M : \tau \cdot \gamma$ can only block on the addition of tags of the form $\tau \odot \tau'$, meaning that with the choice of K made above, the "locks" on shorter tags are always taken before those on longer tags, and thus locks are always taken in a consistent order, an algorithm known to prevent deadlocks.

Intersection and Contraction are implemented in much the same way. When implementing a Contraction, we're basically choosing between multiple possible types for the same term, and when implementing an Intersection, we're simply giving control to a term which can have multiple possible types and multiple possible free variables. The term itself is the same regardless of which of its various possible types it has (thanks to the requirements on Intersection that $\theta_0 \sim \theta_0'$ and that both M are the same). Thus, there are two differences in what happens based on which side of the Contraction/Intersection is used. One is in which tags are present on the types, and thus on how the set of running tags ends up; this is based on which side of the Contraction we arrived from. The other is on which free variables we look at (those from the left-hand side of the intersection, or those from the right-hand side); we can work this out based on what the set of running tags is (because $\theta_0 \# \theta_0'$, we can simply check whether θ_0 or θ_0' is running, and choose the appropriate free variable based on this).

Of course, there would be many cases where this full locking discipline would not be necessary. If our tags represent hard real-time time intervals, for example, the "locks" can be entirely optimized out in the resulting implementation; we can know statically at which time periods a given lock will or won't be held, and in particular, know that there will be no overlap. The only time any of the tags are thus relevant in the final program is when implementing the Intersection rule; it will need to know which free variables to use. This could be implemented via looking at some sort of global clock, then using a lookup table to determine which locks would be effectively held at any given time. An alternative method would be to track the notional state of each lock via observing the control flow of the terms that use it, which has the advantage of not needing any global state. Which method is cheaper would likely depend on the nature and details of the implementation.

The hard real-time setting is not the only setting in which lock contention is impossible. In the next chapter, we consider a situation in which, despite not knowing any specific duration that any term will take to execute, we nonetheless know that there will never be a need to wait.

Chapter 8

SEQUENTIAL CONTRACTION

In the previous chapter, we considered one method of reusing code in an Affine ICA-like type system: synchronizing the program execution such that only one instance of a term would run at a time. This method required a synchronous framework in which two apparently unrelated events could be forced to happen at the same time. For example, synchronized contraction makes a program like $\lambda x.\lambda y.\mathsf{par}(\mathsf{seq}(x)(y))(\mathsf{seq}(y)(x))$ legal, because we can require the first executions of x and y to finish simultaneously.

This technique is probably enough to implement a practically useful system; the functionality of "wait for each of two commands to stop running before continuing" must exist even in an Affine ICA implementation due to the semantics of par, and it is not much of a stretch to imagine generalizing it to types other than com. However, it is a different approach from that taken by languages like bSCI, which do not have a concept of simultaneity. The functionality of the above program is still possible in bSCI, but it looks very different: $\lambda x.\lambda y.\mathsf{seq}(\langle \mathsf{par}(x)(y), \mathsf{par}(y)(x)\rangle)$. The synchronization is now being done by the seq constant, rather than by the type system. bSCI can be considered an asynchronous framework: one in which the only thing affecting timing is the constants. This chapter considers type systems whose purpose is to enforce that a term uses only "sequential contraction", i.e. terms in which it is possible to prove by looking at the term itself that if a subterm is used twice, one instance must happen entirely before the other even in the absence of external synchronization.

Sequential contraction is similar to synchronized contraction, but there is a subtle difference: in synchronized contraction, the type system finds some set of timings for which you never get two simultaneous instances of the same term, whereas in sequential contraction, the type system proves that for all possible sets of timings, you never get two simultaneous instances of the same term. This intuition will become clearer throughout the chapter.

8.1 Basic Syntactic Control of Interference

The basic problem with evaluating terms more than once in a finite-state type system is not that multiple evaluations of the term inherently require an unbounded amount of state; rather, it is that if the term tries to evaluate more than once during the same time period, each of those evaluations will require its own, independent state. In the preceding chapters, we saw various solutions to this problem. Using bounded contraction, and implementing it using linearization, solved the problem by having a separate copy of the term for each evaluation, and thus having a separate place to store the state for each evaluation. An obvious improvement is to re-use the state for terms that are no longer executing; there is no risk of interference between the two evaluations, because one has finished before the other has started. In the previous chapter, we considered implementing this by allocating time intervals to each evaluation; if the intervals do not overlap, then re-using the state is safe. An alternative is to allow contraction only between two variables that, due to the structure of the term that contains them, cannot possibly overlap no matter what the timings are.

To correctly model this type of contraction in a type system, we thus need some concept of whether multiple instances of a term might potentially evaluate at the same time. The language Syntactic Control of Interference was created by Reynolds in [40] for a related purpose: ensuring that there were no circumstances under which there could be interference between multiple identifiers. Interference includes situations in which a variable is written simultaneously from multiple locations, or in which it is written and read at the same time; thus, there is something of a simultaneity restriction existing in the type system already. On the other hand, the language goes to extra effort (introducing "passive types") in order to allow simultaneous reads of the same identifier. This extra power is inappropriate in our context, where even simultaneous reads can cause problems; in a call-by-name semantics such as the one we are using, reading the value of an identifier can require determining the value of an expression, and thus function calls, which might cause multiple concurrent executions of the same term even if the functions have no side effects. As such, it is more useful for our purposes to use Basic Syntactic Control of Interference ("bSCI", introduced by O'Hearn in [34, page 28]; and discussed in Section 3.3.3) instead; it is similar to the full version of SCI, but has no passive types, and thus has a notion of simultaneity that better models our situation. bSCI can be considered to be a type system that basically just adds sequential contraction to Affine ICA.

In order to better compare bSCI with Affine ICA, we give Figure 8.1.1, a presentation of bSCI's type system designed to be as similar as possible to Affine ICA. Most versions of bSCI have three base types (com for commands, exp for integer expressions (typically with no defined range of values), var for integer variables); to keep the type system finite-state, we need a specific, defined range of possible values on our integers (because we only have a finite amount of state to store their values, we need finitely many possible values), and so we introduce a family of exp types, parameterized by their legal values (and just as with Affine ICA, we can then define com as $\mathsf{exp}_{\{0\}}$). var likewise needs to be parameterized to give a range of legal values, but anyway can be defined as $\mathsf{exp}_J \times (\mathsf{exp}_J \to \mathsf{com})$ rather than as a base type of its own, just as we saw with Bounded ICA: a variable can be read (producing an expression), and written to (which requires an expression as an argument and is a command). Besides, we end up with no constants that use it; most versions of bSCI use newvar, assign and deref constants for manipulating their variables, but we wish to be able to use bSCI as a target language for serialization (from SCC), meaning that we need to be able to handle multiple concurrent reads or writes to the variable. Because the constant needs to be changed anyway, we may as well simply use the $\mathsf{newvar}_{J,r,w}$ from Affine ICA directly; it has all the power that is required for this, and the standard definition of newvar is a special case of it.

It can be seen that bSCI is very similar to both Affine ICA, and full ICA; specifically, it has the Product rule from full ICA, but the Application rule of Affine ICA. More interesting, though, is the list of constants. We can clearly use almost all the constants of Affine ICA (par and seq, if, arithmetic operations and base type constants like skip), although we need to go

$$\theta ::= \mathsf{exp}_J \mid \theta \to \theta \mid \theta \times \theta, \text{ where } J \subset \mathbb{N} \text{ is a finite set}$$
$$\Gamma ::= \text{sequence of } x\colon\theta$$
$$\mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$

$$\frac{x \neq y}{x\colon\theta \# y\colon\theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma :: \Gamma'\#\Delta} \qquad \frac{x\colon\theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{x\colon\theta \vdash x\colon\theta}\ \text{Identity}$$

$$\frac{M : \theta \text{ is a constant}}{\vdash M\colon\theta}\ \text{Constant}$$

$$\frac{\Gamma \vdash M\colon\theta \qquad x\#\Gamma}{\Gamma :: x\colon\theta' \vdash M\colon\theta}\ \text{Weakening}$$

$$\frac{\Gamma :: x\colon\theta' :: y\colon\theta'' :: \Delta \vdash M\colon\theta}{\Gamma :: y\colon\theta'' :: x\colon\theta' :: \Delta \vdash M\colon\theta}\ \text{Exchange}$$

$$\frac{\Gamma :: x\colon\theta' :: \Delta \vdash M\colon\theta}{\Gamma :: \Delta \vdash \lambda(x\colon\theta').M\colon(\theta' \to \theta)}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M\colon\theta' \to \theta \qquad \Delta \vdash N\colon\theta' \qquad \Gamma\#\Delta}{\Gamma :: \Delta \vdash MN\colon\theta}\ \text{Application}$$

$$\frac{\Gamma \vdash M\colon\theta \qquad \Gamma \vdash N\colon\theta'}{\Gamma \vdash \langle M,N\rangle\colon(\theta \times \theta')}\ \text{Product}$$

Figure 8.1.1: Basic SCI, formatted like Affine ICA

$$\begin{aligned}
\mathsf{par} &: \mathsf{com} \to (\mathsf{com} \to \mathsf{com})\\
\mathsf{seq}_J &: (\mathsf{com} \times \mathsf{exp}_J) \to \mathsf{exp}_J\\
\mathsf{if}_J &: (\mathsf{exp}_{\{0,1\}} \times \mathsf{exp}_J \times \mathsf{exp}_J) \to \mathsf{exp}_J\\
(\pi_1)_{\theta,\theta'} &: (\theta \times \theta') \to \theta\\
(\pi_2)_{\theta,\theta'} &: (\theta \times \theta') \to \theta'\\
\mathsf{op}_{J,\bullet} &: (\mathsf{exp}_J \times \mathsf{exp}_J) \to \mathsf{exp}_J\\
\mathsf{skip} &: \mathsf{com}\\
j_J &: \mathsf{exp}_J \qquad (j \in \mathbb{N}, j \in J)\\
\mathsf{while} &: (\mathsf{exp}_{\{0,1\}} \times \mathsf{com}) \to \mathsf{com}\\
\mathsf{newvar}_{J,r,w} &: ((\overbrace{\mathsf{exp}_J \times \ldots \times \mathsf{exp}_J}^{r \text{ times}} \times \overbrace{(\mathsf{exp}_J \to \mathsf{com}) \times \ldots \times (\mathsf{exp}_J \to \mathsf{com})}^{w \text{ times}}) \to \mathsf{com}) \to \mathsf{com}
\end{aligned}$$

Figure 8.1.2: Constants used for bSCI

$$
\dfrac{
  \dfrac{}{\vdash \mathsf{uncurry} : (\theta' \to (\theta' \to \theta)) \to ((\theta' \times \theta') \to \theta)}
  \qquad
  \dfrac{\dfrac{\Gamma :: x : \theta' :: y : \theta' \vdash M : \theta}{\Gamma :: x : \theta' \vdash \lambda(y:\theta').M : (\theta' \to \theta)}}{\Gamma \vdash \lambda(x:\theta').\lambda(y:\theta').M : (\theta' \to (\theta' \to \theta))}
}{
  \Gamma \vdash \mathsf{uncurry}(\lambda(x:\theta').\lambda(y:\theta').M) : ((\theta' \otimes \theta') \to \theta)
}
\qquad
\dfrac{\dfrac{}{z:\theta' \vdash z:\theta'} \qquad \dfrac{}{z:\theta' \vdash z:\theta'}}{z:\theta' \vdash \langle z, z\rangle : (\theta' \otimes \theta')}
$$

$$
\Gamma :: z : \theta' \vdash \mathsf{uncurry}(\lambda(x:\theta').\lambda(y:\theta').M)(\langle z,z\rangle) : \theta
$$

Figure 8.1.3: Why bSCI has no uncurry

back to projections π_1 and π_2 rather than uncurry; being unable to read both sides of a tensor is undesirable in Affine ICA, but being unable to read both sides of a product is required in bSCI to prevent general contraction being implemented indirectly through the use of products (Figure 8.1.3 is an example of what goes wrong if uncurry is allowed in bSCI; it uses uncurry to effectively contract arbitrary variables x, y of the same type θ' in an arbitrary term M via constructing a term that's semantically equivalent to $M[z/x][z/y]$). Although in full SCI recursion is defined (by Reynolds in [40, page 43]) in terms of passivity, O'Hearn showed in [34, page 30] that it is in fact possible to add recursion to bSCI. However, we consider only programs that do not use general recursion in this chapter, and thus fix is omitted for the time being; we will consider it later, in Chapter 9. We can now also add a while constant, due to our support for re-evaluation; the various iterations of a while loop can never overlap with each other in time, and thus effectively "contracting" them is safe.

As Reynolds's original paper on SCI ([40]) explains, the type of a function depends on its interference properties; arguments to a function must be non-interfering (i.e. can be evaluated simultaneously), but if we bundle multiple arguments up into one using a product, the various parts of that product may interfere with each other (and thus can only be evaluated sequentially). This can also be observed in the type system: contraction is implicitly allowed between the halves of a product, but not in function application. We can now write down the constants we want for our language; Figure 8.1.2 shows their types.

The most immediately recognizable feature of bSCI (and SCI variants in general) is the difference in the structure of the types of par and of seq. par can evaluate its arguments in parallel, and thus needs to ensure that they don't share. As such, it uses a curried style of application; the type system ensures that a function cannot share with its argument, and so this means that the two arguments to par cannot share with each other (because par is curried, $\mathsf{par}(M)(N)$ means $(\mathsf{par}(M))(N)$, and so a term containing M is being applied to a term containing N, meaning that M and N cannot share). On the other hand, we want it to be legal to sequence a command with itself, because the two executions of the command can never be simultaneous; so seq needs to take a product as its argument ($\lambda x.\mathsf{seq}(\langle x,x\rangle)$ types just fine, because contraction is allowed in product formation).

Although bSCI is known to be finite-state (e.g. as is shown by the compilation to hardware in [19] by Ghica and myself), it turns out to be unsuitable for many of the applications for which a finite-state system might be required: it has many problems, both mathematical and practical. The very largest mathematical problem is that the language does not have a categorical semantics using the usual denotations of tuples using $\otimes$ and functions using $\Rightarrow$. In a symmetric closed category, as would be expected for the semantics of a language with tuples and functions, $[\![(\theta_1 \otimes \theta_2) \to \theta_3]\!]$ is isomorphic to $[\![\theta_1 \to (\theta_2 \to \theta_3)]\!]$ (because $(A \otimes B) \Rightarrow C$ is isomorphic to $A \Rightarrow (B \Rightarrow C)$). This isomorphism does not hold in bSCI; in order for the language's interference protections to work correctly, par may never be allowed to have the type $(\mathsf{com} \times \mathsf{com}) \to \mathsf{com}$, so the denotation of that type cannot be isomorphic to that of the type $\mathsf{com} \to (\mathsf{com} \to \mathsf{com})$ that par actually has. This fact means that most of the existing mathematical results about denotational semantics do not apply to bSCI, meaning that much of the theory has to be worked out from scratch.

Another semantic equivalence that holds in most call-by-name languages, but not bSCI, is $[\![(\lambda x.M)(y)]\!] = [\![M[y/x]]\!]$; this equivalence ("$\beta$-reduction") is actually used as a definition of function application in some operational semantics. This is not entirely fatal in bSCI, because the equivalence does hold from left to right, which is the direction that most semantics care about. However, it fails to hold from right to left; $y : \mathsf{com} \vdash \mathsf{seq}(\langle y,y\rangle) : \mathsf{com}$ types in bSCI, but $y : \mathsf{com} \vdash (\lambda(x : \mathsf{com}).\mathsf{seq}(\langle x,y\rangle))(y) : \mathsf{com}$ does not. This fact has practical implications: refactoring a bSCI program can be very difficult, because it is not generally possible to abstract out specific parts of the program.

This problem has been recognized before as one of the major problems with SCI-like languages. In the original paper on SCI ([40, page 44]), Reynolds mentioned a related but larger problem with full SCI (in which the β-reduction equivalence does not hold in either direction, due to passivity); the same author later suggested a potential solution in [41], which uses intersection types to produce a type system "SCI2". An alternative solution, in terms of passivity and contraction, was presented by O'Hearn et al in [35]; their paper explicitly uses the same term $y\colon\mathsf{com} \vdash (\lambda(x\colon\mathsf{com}).\mathsf{seq}(\langle x,y\rangle))(y)\colon\mathsf{com}$ as an example, concluding that it is "semi-well-typed" in their type system SCIR, but that type system fails to type as many terms as SCI2 does. Both these type systems have goals that diverge somewhat from the main aim of this thesis, however, in that they aim to prevent writes to variables while they are in use elsewhere in the code, rather than preventing simultaneous execution of terms: a term such as $x\colon\mathsf{exp}_J \vdash (\lambda(f\colon\mathsf{exp}_J \to \mathsf{exp}_J).f(f(x)))(\lambda y.\mathsf{op}_{+,J}\langle y,y\rangle)$ does not involve any writing to anything (at least if x is defined to have no side effects), but is invalid in a type system that allows only sequential contraction because the two calls to $\mathsf{op}_{+,J}$ cannot reasonably be contracted (each call needs separate state to remember whether it is currently evaluating its left or right argument).

bSCI has other practical problems, too, including one very major issue: it does not support type inference for concurrency. In the type systems we have looked at so far, such as Bounded ICA, Intersecting ICA, and ISBLL, the special features of the type system are something that the programmer need not worry about. There is no need for a programmer to be aware that x is used twice to be able to write $f(x)(x)$, and when writing $\lambda g.g(\mathsf{skip})$ in such languages, a programmer need not know or care exactly how many times g uses its argument; a type inference algorithm can work all that out for itself. In bSCI, we are not so fortunate. Although it is possible (in fact, quite easy) to write a type inference algorithm for bSCI, the algorithm is no use for working out what can run in parallel and what can share, because that information is already present in the syntax of a program. A programmer must decide whether they mean to write $f(x)(y)$ or $f(\langle x,y\rangle)$ before the type inference algorithm even gets a chance to run, and so the most that such an algorithm can help is to let the programmer know whether they made the right choice or not.

Another impact of this issue is that it is hard to define syntactic sugar for bSCI (i.e. extra syntax added to make a program easier to write, which is defined in terms of other existing syntax). It is not only function applications that can be written in two different, incompatible ways (a parallel version, and a sharing version); the same applies to most other syntactic elements as well. For example, suppose we wanted to add syntactic sugar for a `let` statement, as in `let x = a in M` (which desugars to $(\lambda x.M)(a)$), to a practical programming language based on bSCI. If we want to allow this to apply to more than one variable at a time, `let x = a, y = b in M`, we need to allow two different desugarings: $(\lambda x.\lambda y.M)(a)(b)$ if a and b can run in parallel, or $(\lambda z.M[\pi_1 z/x][\pi_2 z/y])(\langle a, b\rangle)$ if a and b can share; and due to the issues with type inference, these will need different syntax in the original language. This issue came up in practice when I was designing Verity (described in Chapter 10), a practical SCC-based language (which is thus indirectly bSCI-based); the two desugarings are written `let x = a in let y = b in M` and `let x = a and y = b in M` respectively. The programmer has to decide what will run in parallel, and what will share, whenever they write a term; it is hard for a programmer to change their mind later. This split between parallel and sharing syntax thus makes bSCI a hard language to write in (and especially, a hard language in which to maintain existing programs).

The basic result of all of this is that bSCI is very bad at code reuse; something as apparently simple as factoring out common code may make a program fail to type. In the $(\lambda(x:\mathsf{com}).\mathsf{seq}\langle x, y\rangle)(y)$ example, the program failed to type at all; in the case of the `let` statement, it is possible to type a program where y is defined in terms of x, but the program would have to be written as `let x = a in let x = x and y = b in M`, which is far from intuitive or sensible, in order for the resulting bSCI to type correctly. These problems collectively present a serious obstacle to practical use of bSCI.

8.2 Fixing SCI

Although SCI-based languages present practical problems, the basic idea of preventing contraction in parallel contexts but allowing it in sequential contexts is an important one. What we would like is a language which keeps bSCI's advantages via permitting sequential contraction (and no other type of contraction), but avoids as many of its drawbacks as possible. Thus, in this section, we discuss what restrictions exist on such a language.

We first need to identify what the constraints on the language are. We want to be able to support inference of parallelism and sharing, and so we need to ensure that the type system allows the types of par and seq to have the same structure; this is something that is not easy to do in an SCI-like language. We want to make contraction explicit, in order to be able to make reasoning about the language simpler; this is also hard to do in an SCI-like language, because it would require somehow tracking whether terms will eventually be used to form a product or not. And in order to give the semantics a more familiar structure, we need an operation $\otimes$ such that $(\theta_1 \otimes \theta_2) \to \theta_3$ is isomorphic to $\theta_1 \to (\theta_2 \to \theta_3)$.

The last goal here points towards the first change needed from bSCI: instead of using products, we want the language's support for pair-like constructs to be based on tensors. The difficulty is that we now need to distinguish between two sorts of tensor: tensors where the two halves of the tensor can evaluate simultaneously, and those where they cannot. This is the same problem as that indicated by our first goal, though: we need to be able to choose $\theta_1, \theta_2, \theta_1', \theta_2'$ so as to be able to distinguish the types of $\mathsf{par}\colon(\theta_1 \to (\theta_2 \to \mathsf{com}))$ and $\mathsf{seq}\colon(\theta_1' \to (\theta_2' \to \mathsf{com}))$. In other words, we need to move the indication of serial versus parallel from the structure of types onto the types themselves. This also naturally means that contraction will become explicit, achieving all of the goals we set for the language.

Clearly, the types θ_1, θ_2, etc. need to contain all the information that Affine ICA contains; sequential contraction does not allow commands or expressions to forget their identity as commands or expressions. Thus, a good first guess is to follow the pattern in the preceding sections, and use types of the form $\tau \cdot \theta$, where θ is an Affine ICA type and τ is some sort of tag. The most obvious meaning for tags in this particular type system is as a statement about which part of the program is currently running; it means that the state belonging to some term is currently in use, rather than meaningless. $\tau \cdot \theta$, then, means "the type of terms of type θ that can only execute while τ is true". This gives us our contraction rule: $\tau \cdot \theta$ and $\tau' \cdot \theta$ can be safely contracted if and only if τ and τ' are mutually exclusive (in that there is no set of situations which would make τ and τ' true simultaneously), which we can (following the notation in previous chapters) write as $\tau \# \tau'$.

However, this simple definition is complicated to apply in practice. One natural way to define the type of seq would be "$\tau \cdot \mathsf{com} \to \tau' \cdot \mathsf{com} \to (\tau \oplus \tau') \cdot \mathsf{com}$, where $\tau \# \tau'$". But consider what happens when this definition is used with the following term:

$$x_1 :: x_2 :: y_1 :: y_2 \vdash \mathsf{par}(\mathsf{seq}(x_1)(x_2))(\mathsf{seq}(y_1)(y_2))$$

Nothing prevents x_1 being contracted with y_2 and y_1 with x_2, which doesn't fit with the semantics we'd expect for par. (In fact, the resulting type system is basically call-by-name SBLL, and in the previous chapter, we saw that at least with respect to finite-state type systems, SBLL variants are mostly useful for synchronized contraction.) Producing synchronized rather than sequential composition is a sign that we have the wrong quantifiers. More specifically, the problem is that seq does not *respect* a condition on the state of the program (i.e. the sequential composition $\mathsf{seq}(x)(y)$ does not mean "do x during a certain given time period, and y during a certain other given time period that does not overlap it"); rather, it *produces* a condition on the state of the program (i.e. $\mathsf{seq}(x)(y)$ means "execute x and y, such that there is a condition on the state of the program that is true while x is executing but false while y is executing"). This differs from our naive definition of the type of seq in the quantifiers; the type of seq (in informal notation) is not $\forall \tau, \tau'. \tau \cdot \mathsf{com} \to (\tau' \cdot \mathsf{com} \to (\tau \vee \tau') \cdot \mathsf{com})$, but something more like $\forall \tau''. \exists \tau \oplus \tau' = \tau''. \tau \cdot \mathsf{com} \to (\tau' \cdot \mathsf{com} \to \tau'' \cdot \mathsf{com})$.

The usual method of encoding existentially-quantified types in a type system (e.g. as Läufer explains in [31, page 46]) is to replace existential quantifiers in covariant positions with universal quantifiers in contravariant positions. For example, removing the side-conditions from the type above gives $\forall \tau''. \exists \tau. \exists \tau'. \tau \cdot \mathsf{com} \to (\tau' \cdot \mathsf{com} \to \tau'' \cdot \mathsf{com})$, which is equivalent to $\forall \tau''. (\forall \tau. \tau \cdot \mathsf{com}) \to ((\forall \tau'. \tau' \cdot \mathsf{com}) \to \tau'' \cdot \mathsf{com})$. This sort of type can easily be represented in variants of System F, so is far from unprecedented. Sadly, the $\tau \oplus \tau' = \tau''$ requirement prevents this transformation applying directly. In order to be able to encode it with only universal quantifiers, we would need τ and τ' to be grouped together somehow, as in $\forall \tau''. (\forall \tau \oplus \tau' = \tau''. \tau \cdot \mathsf{com} \otimes$

$\tau' \cdot \mathsf{com}) \to \tau'' \cdot \mathsf{com}$. In other words, in type systems designed to allow sequential contraction and that do not use existential types, there is a genuine difference between curried and uncurried functions; because curried functions associate to the right rather than the left, there are no contravariant positions containing both τ' and τ'' at which it would be possible to place a side condition that $\tau \oplus \tau' = \tau''$ on a universal quantifier (and although appropriate covariant positions exist, a quantifier in such a position would have to be existential rather than universal to have the meaning we want).

Thus, in order to keep our function arrow adjoint to our tensor, we need some other encoding. We can exploit the fact that the types of quantification that are used in the type system are quite limited. Every term has a universal quantifier on the type of its return value; because this is true of every term, we can make this a rule of the type system, rather than part of the type itself, and this greatly simplifies the types (e.g. par now has a type of $1 \cdot \mathsf{com} \to 1 \cdot \mathsf{com} \to 1 \cdot \mathsf{com}$ rather than $\forall \tau''.(\forall \tau.\tau \cdot \mathsf{com}) \to (\forall \tau'.\tau' \cdot \mathsf{com}) \to \tau'' \cdot \mathsf{com}$). The only remaining uses of quantifiers are for terms that guarantee that their arguments are not used in parallel (e.g. seq now looks something like $\exists(\tau \# \overline{\tau}).(\tau \cdot \mathsf{com} \to \overline{\tau} \cdot \mathsf{com} \to 1 \cdot \mathsf{com})$), and for higher-order terms that want to require that their arguments do not use their arguments in parallel (such as $\lambda f.((\lambda x.f(x)(x))(\mathsf{skip}))$, which has a type of $(\exists(\tau \# \overline{\tau}).(\tau \cdot \mathsf{com} \to \overline{\tau} \cdot \mathsf{com} \to 1 \cdot \mathsf{com})) \to 1 \cdot \mathsf{com}$ or equivalently $\forall(\tau \# \overline{\tau}).((\tau \cdot \mathsf{com} \to \overline{\tau} \cdot \mathsf{com} \to 1 \cdot \mathsf{com}) \to 1 \cdot \mathsf{com})$.

Our observation that sequential and synchronized contraction are the same except for quantifiers points to a solution. In Section 7.4, we saw an instance of ISBLL that had synchronized contraction and local timescales. An interesting property of this instance is that given any element υ of $\mathbb{A}$, consistently replacing it with any other element of $\mathbb{A}$ does not prevent the program from typing (as it can only add $\tau \# \tau'$ relations; any such relations that existed before the replacement will continue to exist afterwards). This has a useful application for constants such as $\mathsf{seq}_{\upsilon,J} : \upsilon^- \cdot \mathsf{com} \to (\upsilon^+ \cdot \mathsf{exp}_J \to \mathsf{exp}_J)$: if a term containing $\mathsf{seq}_{\upsilon,J}$ types correctly in this instance of ISBLL and υ does not appear in any other use of a constant, then the term will continue to type correctly if, for any υ', we replace $\mathsf{seq}_{\upsilon,J}$ with $\mathsf{seq}_{\upsilon',J}$. This gives us the ability to handle existential quantifiers that we need: so long as all elements of $\mathbb{A}$ that parameterize constants happen to be unique, existential and universal quantification are equivalent in this instance of ISBLL.

This means that, instead of bSCI, we can represent sequential contraction using ISBLL with a side condition. We consider this type system in the next section.

8.3 Tagged Control of Interference

The previous section suggested that to model sequential composition, we can use the instance of ISBLL in Section 7.4, together with a requirement that all the υ used as subscripts to constants are unique. This type system, Tagged Control of Interference (or TCI), is only a minor change to ISBLL, and shown in Figure 8.3.1; the only changes are the addition of a second context, that lists the υ that have been used by constants (in order to ensure that each υ is only used by one constant), and that we have now fixed K to be the specific semigroup discussed in Section 7.4. The constants are likewise defined by analogy with ISBLL, and shown in Figure 8.3.2 (where the sum in newvar is defined the same way as with ISBLL); the sequential constants now have a υ in the Ψ context, meaning that two different sequential constants are incapable of synchronizing against each other.

We consider two examples of how this type system works. Our first example, which we have frequently used as an example of synchronized contraction, is $\lambda x.\lambda y.\mathsf{par}(\mathsf{seq}(x)(y))(\mathsf{seq}(y)(x))$; we expect this not to type correctly in TCI, and indeed, the term fails to type. Figure 8.3.3a shows an attempt at deriving this term, using unknowns for the tags; the production marked [1] is only legal if $\upsilon_1 \neq \upsilon_2$ (so that $\{\upsilon_1\}\#\{\upsilon_2\}$), meaning that x_1 cannot be contracted with x_2 nor y_1 with y_2 because they would require $\upsilon_1^-\#\upsilon_2^+$ and $\upsilon_1^+\#\upsilon_2^-$ respectively (both of which can only be true if $\upsilon_1 = \upsilon_2$). Meanwhile, Figure 8.3.3b shows how $(\lambda x.\mathsf{seq}(x)(y))(y)$ types correctly in TCI, even though the bSCI equivalent $(\lambda x.\mathsf{seq}(\langle x,y\rangle))(y)$ fails to type in bSCI. As expected, it types with the exact same type that $\mathsf{seq}(y)(y)$ would.

Ideally, a type system like TCI should type all terms that use only sequential contraction, but

$$\gamma ::= \mathsf{exp}_J \mid \theta \to \theta \mid \theta + \theta \mid \theta \otimes \theta, \text{ where } J \subset \mathbb{N} \text{ is a finite set} \qquad \mathsf{com} \triangleq \mathsf{exp}_{\{0\}}$$
$$\theta ::= \tau \cdot \gamma$$
$$\tau ::= \varepsilon \mid \upsilon^{+}\tau \mid \upsilon^{-}\tau, \text{ where } \upsilon \in \mathbb{A}$$
$$\Gamma ::= \text{sequence of } x\colon\theta$$
$$\Psi ::= \text{subset of } \mathbb{A} \qquad \Psi\#\Psi' \text{ iff } \Psi \cap \Psi' = \emptyset$$
$$\tau \cdot \varepsilon \triangleq \varepsilon \quad \tau \cdot (x\colon\tau' \cdot \gamma :: \Gamma) \triangleq x\colon(\tau \odot \tau') \cdot \gamma :: (\tau \cdot \Gamma)$$

$$\frac{x \neq y}{x\colon\theta\#y\colon\theta'} \qquad \frac{\Gamma\#\Delta \qquad \Gamma'\#\Delta}{\Gamma :: \Gamma'\#\Delta} \qquad \frac{x\colon\theta\#\Gamma}{x\#\Gamma}$$

$$\frac{}{\tau \cdot \mathsf{exp}_J \sim \tau' \cdot \mathsf{exp}_J} \qquad \frac{\theta_1 \sim \theta_2 \qquad \theta_1' \sim \theta_2'}{\tau \cdot (\theta_1 \otimes \theta_2) \sim \tau' \cdot (\theta_1' \otimes \theta_2')} \qquad \frac{\theta_1 \sim \theta_2 \qquad \theta_1' \sim \theta_2'}{\tau_1 \cdot (\theta_1' \to \theta_1) \sim \tau_2 \cdot (\theta_2' \to \theta_2)}$$

$$\frac{\theta_1 \sim \theta_2 \qquad \theta_1' \sim \theta_2}{\tau \cdot (\theta_1 + \theta_1') \sim \theta_2} \qquad \frac{\theta_1 \sim \theta_2 \qquad \theta_1 \sim \theta_2'}{\theta_1 \sim \tau \cdot (\theta_2 + \theta_2')}$$

$$\frac{\{\tau_1', \tau_2'\} = \{\upsilon^{+}, \upsilon^{-}\}}{\tau\tau_1'\tau_1 \cdot \gamma_1\#\tau\tau_2'\tau_2 \cdot \gamma_2} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1'\#\theta_2}{\varepsilon \cdot (\theta_1 + \theta_1')\#\theta_2} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1\#\theta_2'}{\theta_1\#\varepsilon \cdot (\theta_2 + \theta_2')}$$

$$\frac{\theta_1\#\theta_2 \qquad \theta_1'\#\theta_2}{\varepsilon \cdot (\theta_1 \otimes \theta_1')\#\theta_2} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1\#\theta_2'}{\theta_1\#\varepsilon \cdot (\theta_2 \otimes \theta_2')} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1'\#\theta_2}{\varepsilon \cdot (\theta_1' \to \theta_1)\#\theta_2} \qquad \frac{\theta_1\#\theta_2 \qquad \theta_1\#\theta_2'}{\theta_1\#\varepsilon \cdot (\theta_2' \to \theta_2)}$$

$$\frac{}{\emptyset \mid x\colon\theta \vdash x\colon\theta}\ \text{Identity} \qquad \frac{M\colon\Psi \mid \theta \text{ is a constant}}{\Psi \mid \varepsilon \vdash M\colon\theta}\ \text{Constant}$$

$$\frac{\Psi \mid \Gamma \vdash M\colon\theta \qquad x\#\Gamma}{\Psi \mid \Gamma :: x\colon\theta' \vdash M\colon\theta}\ \text{Weakening} \qquad \frac{\Psi \mid \Gamma :: x\colon\theta' :: y\colon\theta'' :: \Delta \vdash M\colon\theta}{\Psi \mid \Gamma :: y\colon\theta'' :: x\colon\theta' :: \Delta \vdash M\colon\theta}\ \text{Exchange}$$

$$\frac{\Psi \mid \Gamma :: x\colon\theta' :: \Delta \vdash M\colon\theta}{\Psi \mid \Gamma :: \Delta \vdash \lambda(x\colon\theta').M\colon\varepsilon \cdot (\theta' \to \theta)}\ \text{Abstraction}$$

$$\frac{\Psi \mid \Gamma \vdash M\colon\tau \cdot \gamma}{\Psi \mid \tau' \cdot \Gamma \vdash M\colon(\tau' \odot \tau) \cdot \gamma}\ \text{Tagging}$$

$$\frac{\Psi \mid \Gamma \vdash M\colon\varepsilon \cdot (\theta' \to \theta) \qquad \Psi' \mid \Delta \vdash N\colon\theta' \qquad \Gamma\#\Delta \qquad \Psi\#\Psi'}{\Psi \cup \Psi' \mid \Gamma :: \Delta \vdash MN\colon\theta}\ \text{Application}$$

$$\frac{\Psi \mid \Gamma \vdash M\colon\theta \qquad \Psi' \mid \Delta \vdash N\colon\theta' \qquad \Gamma\#\Delta \qquad \Psi\#\Psi'}{\Psi \cup \Psi' \mid \Gamma :: \Delta \vdash M \otimes N\colon\varepsilon \cdot (\theta \otimes \theta')}\ \text{Tensor}$$

$$\frac{\begin{array}{c}\Psi \mid x_1\colon\theta_1 :: \ldots :: x_j\colon\theta_j \vdash M\colon\theta_0 \\ \Psi' \mid x_1\colon\theta_1' :: \ldots :: x_j\colon\theta_j' \vdash M\colon\theta_0'\end{array} \qquad \bigwedge_{k=0}^{j} \theta_k \sim \theta_k' \qquad \bigwedge_{k=0}^{j} \theta_k\#\theta_k' \qquad \Psi\#\Psi'}{\Psi \cup \Psi' \mid x_1\colon\varepsilon \cdot (\theta_1 + \theta_1') :: x_2\colon\varepsilon \cdot (\theta_2 + \theta_2') :: \ldots :: x_j\colon\varepsilon \cdot (\theta_j + \theta_j') \vdash M\colon\varepsilon \cdot (\theta_0 + \theta_0')}\ \text{Intersection}$$

$$\frac{\Psi \mid \Gamma :: x\colon\theta' :: y\colon\theta'' \vdash M\colon\theta \qquad \theta \sim \theta' \qquad \theta\#\theta'}{\Psi \mid \Gamma :: x\colon\varepsilon \cdot (\theta' + \theta'') \vdash M[x/y]\colon\theta}\ \text{Contraction}$$

Figure 8.3.1: Tagged Control of Interference

$$\begin{aligned}
\mathsf{par} &: \emptyset \mid \varepsilon \cdot (\varepsilon \cdot \mathsf{com} \to \varepsilon \cdot (\varepsilon \cdot \mathsf{com} \to \varepsilon \cdot \mathsf{com}))\\
\mathsf{seq}_{J,\upsilon} &: \upsilon \mid \varepsilon \cdot (\upsilon^{-} \cdot \mathsf{com} \to \varepsilon \cdot (\upsilon^{+} \cdot \mathsf{exp}_J \to \varepsilon \cdot \mathsf{exp}_J))\\
\mathsf{if}_{J,\upsilon} &: \upsilon \mid \varepsilon \cdot (\upsilon^{-} \cdot \mathsf{exp}_{\{0,1\}} \to \varepsilon \cdot (\upsilon^{+} \cdot \mathsf{exp}_J \to \varepsilon \cdot (\upsilon^{+} \cdot \mathsf{exp}_J \to \varepsilon \cdot \mathsf{exp}_J)))\\
\mathsf{uncurry}_{\theta',\theta'',\gamma} &: \emptyset \mid \varepsilon \cdot (\varepsilon \cdot (\theta' \to \varepsilon \cdot (\theta'' \to \theta)) \to \varepsilon \cdot (\varepsilon \cdot (\theta' \otimes \theta'') \to \theta))\\
\mathsf{op}_{J,\bullet,\upsilon} &: \upsilon \mid \varepsilon \cdot (\upsilon^{-} \cdot \mathsf{exp}_J \to \varepsilon \cdot (\upsilon^{+} \cdot \mathsf{exp}_J \to \varepsilon \cdot \mathsf{exp}_J))\\
\mathsf{skip} &: \emptyset \mid \varepsilon \cdot \mathsf{com}\\
j_J &: \emptyset \mid \varepsilon \cdot \mathsf{exp}_J \qquad (j \in \mathbb{N}, j \in J)\\
\mathsf{while}_{\upsilon} &: \upsilon \mid \varepsilon \cdot (\upsilon^{-} \cdot \mathsf{exp}_{\{0,1\}} \to \varepsilon \cdot (\upsilon^{+} \cdot \mathsf{com} \to \varepsilon \cdot \mathsf{com}))\\
\mathsf{newvar}_{J,A} &: \emptyset \mid \varepsilon \cdot (\varepsilon \cdot (\textstyle\sum_{\langle \tau,\tau' \rangle \in A} \varepsilon \cdot (\tau \cdot \mathsf{exp}_J \otimes \tau' \cdot (\varepsilon \cdot \mathsf{exp}_J \to \varepsilon \cdot \mathsf{com})) \to \varepsilon \cdot \mathsf{com}) \to \varepsilon \cdot \mathsf{com})\\
&\qquad (A \subset K \times K, A \text{ is finite})
\end{aligned}$$

Figure 8.3.2: Constants used for TCI

not those that require other forms of contraction. However, we have a bit of a circular definition here: we defined sequential contraction as contraction between variables that cannot execute simultaneously due to the structure of the term, which is both syntactic and informal. TCI can be seen as a formalization of this notion, meaning that an attempt to prove that TCI admits all sequentially contracting terms is vacuously true. A proof of a result along these lines would instead have to be based on a semantics (which could perhaps be produced via linearization to Affine ICA, via the use of the fact that TCI is a subset of ISBLL which is a subset of Intersecting ICA), or perhaps via showing that all bSCI terms type in TCI (nontrivial due to the differences in the constants, and which would be a weaker result).

It is also worth thinking about what an implementation of TCI looks like. When implementing ISBLL, we considered the tags to effectively work as a locking mechanism; while a term was running, it locked out any terms with contradictory tags from running. As TCI is ISBLL with a side condition, this implementation would still work. However, the side condition gives us extra knowledge: terms with contradictory tags can never run at the same time, because contradictory tags can only be introduced by Weakening (in which case the term never runs at all), or via constants such as seq or while whose purpose is to run things sequentially. Thus, there

For brevity, “$\tau \cdot \mathsf{com}$” is abbreviated as “$\overline{\tau}$” in these derivations, and $\varepsilon\cdot$ prefixes are omitted.

$$\dfrac{\dfrac{\dfrac{}{\{\upsilon_1\} \mid \varepsilon \vdash \mathsf{seq}_{\{0\},\upsilon_1} : \overline{\upsilon_1^-} \to (\overline{\upsilon_1^+} \to \overline{\varepsilon})} \qquad \dfrac{}{\emptyset \mid x_1 : \overline{\upsilon_1^-} \vdash x_1 : \overline{\upsilon_1^-}}}{\{\upsilon_1\} \mid x_1 : \overline{\upsilon_1^-} \vdash \mathsf{seq}_{\{0\},\upsilon_1}(x_1) : \overline{\upsilon_1^+} \to \overline{\varepsilon}} \qquad \dfrac{}{\emptyset \mid y_1 : \overline{\upsilon_1^+} \vdash y_1 : \overline{\upsilon_1^+}}}{\{\upsilon_1\} \mid x_1 : \overline{\upsilon_1^-} :: y_1 : \overline{\upsilon_1^+} \vdash \mathsf{seq}_{\{0\},\upsilon_1}(x_1)(y_1) : \overline{\varepsilon}}$$

$$\dfrac{\dfrac{\dfrac{}{\{\upsilon_2\} \mid \varepsilon \vdash \mathsf{seq}_{\{0\},\upsilon_2} : \overline{\upsilon_2^-} \to (\overline{\upsilon_2^+} \to \overline{\varepsilon})} \qquad \dfrac{}{\emptyset \mid y_2 : \overline{\upsilon_2^-} \vdash y_2 : \overline{\upsilon_2^-}}}{\{\upsilon_2\} \mid y_2 : \overline{\upsilon_2^-} \vdash \mathsf{seq}_{\{0\},\upsilon_2}(y_2) : \overline{\upsilon_2^+} \to \overline{\varepsilon}} \qquad \dfrac{}{\emptyset \mid x_2 : \overline{\upsilon_2^+} \vdash x_2 : \overline{\upsilon_2^+}}}{\{\upsilon_2\} \mid y_2 : \overline{\upsilon_2^-} :: x_2 : \overline{\upsilon_2^+} \vdash \mathsf{seq}_{\{0\},\upsilon_2}(y_2)(x_2) : \overline{\varepsilon}}$$

$$\dfrac{\dfrac{\dfrac{}{\emptyset \mid \varepsilon \vdash \mathsf{par} : \overline{\varepsilon} \to (\overline{\varepsilon} \to \overline{\varepsilon})} \qquad \{\upsilon_1\} \mid \varepsilon \vdash \mathsf{seq}_{\{0\},\upsilon_1} : \overline{\upsilon_1^-} \to (\overline{\upsilon_1^+} \to \overline{\varepsilon})}{\{\upsilon_1\} \mid x_1 : \overline{\upsilon_1^-} :: y_1 : \overline{\upsilon_1^+} \vdash \mathsf{par}(\mathsf{seq}_{\{0\},\upsilon_1}(x_1)(y_1)) : \overline{\varepsilon} \to \overline{\varepsilon}} \qquad \{\upsilon_2\} \mid y_2 : \overline{\upsilon_2^-} :: x_2 : \overline{\upsilon_2^+} \vdash \mathsf{seq}_{\{0\},\upsilon_2}(y_2)(x_2) : \overline{\varepsilon}}{\{\upsilon_1, \upsilon_2\} \mid x_1 : \overline{\upsilon_1^-} :: y_1 : \overline{\upsilon_1^+} :: y_2 : \overline{\upsilon_2^-} :: x_2 : \overline{\upsilon_2^+} \vdash_t \mathsf{par}(\mathsf{seq}_{\{0\},\upsilon_1}(x_1)(y_1))(\mathsf{seq}_{\{0\},\upsilon_2}(y_2)(x_2)) : \overline{\varepsilon}}\,[1]$$

(a) $\lambda x.\lambda y.\mathsf{par}(\mathsf{seq}(x)(y))(\mathsf{seq}(y)(x))$ is not derivable

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathsf{u}_1\} \mid \varepsilon \vdash \mathsf{seq}_{\{0\},\mathsf{u}_1} : \overline{\mathsf{u}_1^-} \to (\overline{\mathsf{u}_1^+} \to \overline{\varepsilon})} \qquad \dfrac{}{\emptyset \mid x : \overline{\mathsf{u}_1^-} \vdash x : \overline{\mathsf{u}_1^-}}}{\{\mathsf{u}_1\} \mid x : \overline{\mathsf{u}_1^-} \vdash \mathsf{seq}_{\{0\},\mathsf{u}_1}(x) : \overline{\mathsf{u}_1^+} \to \overline{\varepsilon}} \qquad \dfrac{}{\emptyset \mid y : \overline{\mathsf{u}_1^+} \vdash y : \overline{\mathsf{u}_1^+}}}{\{\mathsf{u}_1\} \mid x : \overline{\mathsf{u}_1^-} :: y : \overline{\mathsf{u}_1^+} \vdash \mathsf{seq}_{\{0\},\mathsf{u}_1}(x)(y) : \overline{\varepsilon}}}{\{\mathsf{u}_1\} \mid y : \overline{\mathsf{u}_1^+} \vdash \lambda(x : \overline{\mathsf{u}_1^-}).\mathsf{seq}_{\{0\},\mathsf{u}_1}(x)(y) : \overline{\mathsf{u}_1^-} \to \overline{\varepsilon}} \qquad \dfrac{}{\emptyset \mid z : \overline{\mathsf{u}_1^-} \vdash z : \overline{\mathsf{u}_1^-}}}{\dfrac{\{\mathsf{u}_1\} \mid y : \overline{\mathsf{u}_1^+} :: z : \overline{\mathsf{u}_1^-} \vdash (\lambda(x : \overline{\mathsf{u}_1^-}).\mathsf{seq}_{\{0\},\mathsf{u}_1}(x)(y))(z) : \overline{\varepsilon}}{\{\mathsf{u}_1\} \mid y : (\overline{\mathsf{u}_1^+} + \overline{\mathsf{u}_1^-}) \vdash (\lambda(x : \overline{\mathsf{u}_1^-}).\mathsf{seq}_{\{0\},\mathsf{u}_1}(x)(y))(y) : \overline{\varepsilon}}}$$

(b) $(\lambda x.\mathsf{seq}(x)(y))(y)$ is derivable

Figure 8.3.3: Type derivations in TCI

is never any need for blocking; rather, all that is necessary is to track which side of each seq, while, if or op is running at any given time (so that the Intersection rule can match up a term with its appropriate set of free variables at any given moment). The obvious implementation of this is for each element υ of Ψ to be associated with one bit (i.e. binary digit) of state, which is set and cleared by the constant that "owns" that element, and for the implementation of Intersection to look at these bits.

There is one problem with the above approach. In a term like $\Psi \mid x : \tau \cdot \mathsf{com} :: y : \tau' \cdot \mathsf{com} \vdash \mathsf{seq}(\mathsf{seq}(x)(y))(\mathsf{seq}(y)(x)) : \mathsf{com}$, TCI as defined will place three elements into Ψ, one for each use of seq. However, it is clear that the second and third uses of seq can share their state, as they can never be executing at the same time; $(\lambda z.\mathsf{seq}(z(x)(y))(z(y)(x)))(\mathsf{seq})$ is also well-typed in TCI, and only requires a two-element Ψ. Thus, an implementation would ideally want to reduce the amount of state necessary via merging some of it. It seems likely that a "more efficient" TCI-like language exists in which contraction inside Ψ is possible; however, Ψ would have to be substantially more complex for this to be correct (there is clearly not enough information inside Ψ at the moment to know whether contraction would be safe). This may be an interesting direction for future research.

Chapter 9

RECURSION

As noted by Ghica, Murawski and Ong [18], there are two main reasons why equivalence of ICA terms generally is undecidable. One is the fact that ICA has no contraction bounds; this was the main topic of Chapter 6. The other is that ICA has a general recursion constant $\mathsf{fix} : (\theta \to \theta) \to \theta$ (intended to have its semantics defined such that $\mathsf{fix}(M) \equiv M(\mathsf{fix}(M))$), which is intended for the construction of infinite-state programs. There are numerous examples that show that fix in ICA cannot be finite-state. One such example is $x : \mathsf{exp}_{\{0,1\}} :: y : \mathsf{com} \vdash \mathsf{fix}(\lambda(z : \mathsf{com}).\mathsf{if}(x)(\mathsf{seq}(z)(y))(\mathsf{skip})) : \mathsf{com}$, which repeatedly evaluates x until it evaluates to 0, then runs y as many times as it evaluated to 1, and thus is effectively capable of storing an unbounded integer (requiring infinitely many states).

Recursion itself is generally a useful concept, however. In some formalizations of programming, any sort of loop is recursion: as an example, while could be defined as $\lambda(x : \mathsf{exp}_{\{0,1\}}).\lambda(y : \mathsf{com}).\mathsf{fix}(\lambda(z : \mathsf{com}).\mathsf{if}(x)(\mathsf{seq}(y)(z))(\mathsf{skip})) : \mathsf{com}$. This is almost identical to the previous example – apart from the lambdas on x and y, the only difference is that the arguments to seq have been exchanged – but it is now finite-state. In this case, the recursion is finite-state due to a special case known as *tail recursion* in which a term's recursive call can happen at most once, and only after all other calculations performed by that term have finished. This means that any state used by the term itself can be discarded just before the recursive call happens. In a call-by-name setting, tail recursion is only possible if there are no arguments to the recursive call (except for those given indirectly via free variables, which are necessarily the same at each recursive level; newvar is useful here because the variable can be the same variable even though its value changes). This is because it is impossible in general to evaluate the argument given to a tail-recursive call once the call has happened; the state used by the argument has already disappeared by that point. Thus, tail-recursion can only occur on base types, meaning that it can be implemented using while (sufficient for tail-recursion on com), plus newvar (needed for tail-recursion on exp, as the return value needs to be stored somewhere).

In this chapter, therefore, we look at what other sorts of recursion might potentially make sense in a finite-state environment.

9.1 Affine recursion

If general recursion is clearly infinite-state, and tail recursion is already present in our languages, is there some more powerful middle ground we could adopt? Various suggestions have been made in the literature. Reynolds's original paper on SCI ([40, page 43]) defines recursion in terms of passivity; this definition perhaps disallows more than it should, though (as Reynolds mentions in [41, page 18]). Another possibility is the concept of *affine recursion*, which was defined by O'Hearn as part of the definition of bSCI ([34, pages 29, 30]), and is shown in Figure 9.1.1a.

This rule is perhaps a little unfamiliar, in that fix cannot be a constant as in ICA, but is instead a syntactic element (Recursion cannot be separated into Constant, Abstraction and Application, like most rules that only define a constant can, due to excluding all variables but x from the context). The extra restriction is necessary because with a nonempty Γ, free variables might be used from different levels of recursion simultaneously (thus forcing any state in the implementations of those variables to be shared): a simple example of this is $y\!:\!\mathsf{com} \vdash \mathsf{fix}\, \lambda x.(\mathsf{par}(x)(y))$, which runs infinitely many copies of y in parallel (and does not type in bSCI plus affine recursion due to the free variable y in the context).

There are two problems with affine recursion. The larger problem is related to constants, and we consider that in the next section. First, we look at a problem that is perhaps smaller, but that often comes up in practice: the restriction on free variables that is imposed by affine recursion is one that tends to cause a lot of repetition in a program. Chapter 10 discusses a practical programming language Verity based on SCC and bSCI, and which currently uses this

$$\frac{x : \theta \vdash M : \theta}{\vdash \mathsf{fix}\,\lambda x.M : \theta}\ \text{Recursion}$$

(a) The recursion rule of bSCI

$$\frac{y_1 : j_1 \cdot \theta_1 :: y_2 : j_2 \cdot \theta_2 :: \ldots :: y_k : j_k \cdot \theta_k :: x : j \cdot \theta \vdash M : \theta \qquad \exists j'.j \oplus j' = \mathbf{1}}{y_1 : (j_1 \oplus j) \cdot \theta_1 :: y_2 : (j_2 \oplus j) \cdot \theta_2 :: \ldots :: y_k : (j_k \oplus j) \cdot \theta_k \vdash \mathsf{fix}\,\lambda x.M : \theta}\ \text{Recursion}$$

(b) An appropriate recursion rule for call-by-name SBLL

Figure 9.1.1: Definitions of the affine recursion rule

affine recursion rule. Attempts to write recursive programs in Verity have been discovered to interact badly with attempts to abstract out common code: the need for an empty context means that common subroutines cannot be defined outside a recursion, then directly accessed from inside it. There is a frequently available workaround, in that $y : \theta' \vdash \mathsf{fix}\,\lambda x.M : \theta$ can be rewritten as $y : \theta' \vdash (\mathsf{fix}\,\lambda x.(\lambda y' : \theta').M[x(y')/x][y'/y])(y) : \theta$ (this is a standard technique, normally found as one of the steps in the "lambda lifting" implementation of block-scoped languages), which implies that $y : \theta' \vdash \mathsf{fix}\,\lambda x.M : \theta$ should be considered well-typed whenever $x : (\theta' \to \theta) :: y : \theta' \vdash M[x(y)/x] : \theta$ is. (This correctly rejects terms like $y : \mathsf{com} \vdash \mathsf{fix}\,\lambda x.(\mathsf{par}(x)(y))$, because $x : (\mathsf{com} \to \mathsf{com}) :: y : \mathsf{com} \vdash \mathsf{par}(x(y))(y)) : \mathsf{com}$ is clearly not derivable in bSCI.)

This "practical" variant of affine recursion is much more convenient to work with as a programmer, but much more complex from the point of view of the type system; there does not seem to be an obvious method to express $x : (\theta' \to \theta) :: y : \theta' \vdash M[x(y)/x] : \theta$ without the substitution in bSCI. However, the condition becomes much simpler in type systems in which beta-reduction is an equivalence (and thus we can replace substitutions with lambdas), such as call-by-name SBLL. In call-by-name SBLL, we could derive the condition by starting with something along the lines of $x : j'' \cdot (j' \cdot \theta' \multimap \theta) :: y : j \cdot \theta' \vdash M[x(y)/x] : \theta$ and then solving for the various j, but this quickly becomes hard to follow. A much simpler way to look at the problem is that in $y : \theta' \vdash \mathsf{fix}\,\lambda x.M : \theta$, any uses of y in M will also be needed by the recursive call to x. This gives us the affine recursion rule shown in Figure 9.1.1b. (The side condition on j is necessary to prevent non-affine recursions in the case where $k = 0$.)

This rule is comparatively new, and thus its implications have not yet been fully studied.

(Most of the research in this chapter has been implemented – Verity, discussed in Chapter 10, contains bSCI's recursion rule – but Verity is based on SCC, rather than an SBLL or TCI variant, and so this rule cannot be directly applied.) However, it seems highly likely that it would avoid the need for lambda lifting in recursion in practical programs, something that has been a source of irritation when Verity programming in the past. At the very least, something along these lines seems like it will be useful for any future programming language with affine typing.

9.2 Recursion with stateful constants

bSCI's recursion rule $\vdash \mathsf{fix}\,\lambda x.M : \theta$ serves its purpose for bSCI, in that it prevents unwanted interference in concurrent contexts. However, if we are using bSCI for its synchronized contraction properties, rather than its interference properties, we run into a problem: M itself may need some internal state to implement. A good example is:

$$y : \mathsf{exp}_{\{0,1\}} :: (z : \mathsf{exp}_{\{0,1\}} \to \mathsf{com}) \vdash$$
$$(\mathsf{fix}\,\lambda x.\lambda y.\lambda z.\mathsf{if}(y)(\mathsf{seq}(\langle \mathsf{seq}(\langle x(y)(z), z(0)\rangle), \mathsf{seq}(\langle x(y)(z), z(1)\rangle)\rangle))(\mathsf{skip}))(y)(z)$$

This may be easier to read in Verity notation: (`fix \x.\y.\z.if y then {x y z;z 0;x y z;z 1} else skip)(y)(z)`). The term clearly requires an infinite amount of internal storage (the number of bits of storage is proportional to the recursion depth, i.e. the number of possible states is exponential in the recursion depth), and yet it types in bSCI just fine. This is not a problem with bSCI for its originally intended use – there's no interference between terms going on here, as the program has no concurrency – but it is a problem if we want a finite-state type system. There are two ways to think about why this term needs infinite state. A contraction-focused point of view is that one bit of state is needed to remember which side of the contraction on x is being executed at each recursion depth, so that when a call to x returns, execution can continue with the correct argument to z. Chapter 7 gives an alternate interpretation of the same phenomenon: seq requires state in order that it can distinguish between uses of free variables from its first argument and from its second argument.

Essentially the same problem came up in SBLL; one solution we considered there was to add constant tracking to the type system, in which constants were considered to act like free variables. Applying this solution directly would imply that just as fix disallows free variables in the context, it should also disallow constants in M. This solution does actually work, in the sense that it reduces recursion to being only finite-state; however, it is clearly an excessively draconian restriction that leaves recursion almost useless.

We can extend this line of thought a little further. If constants act like free variables, then if we could apply contraction to them between different recursion depths, we could effectively lambda-lift them right out of the recursion. When we combine this with the recursion rule in Figure 9.1.1b, we discover that if a constant takes a recursive call as an argument (for example, in $\mathsf{fix}\,\lambda x.\mathsf{seq}(\mathsf{skip})(x)$), then we find that (in this example), the type of $\mathsf{seq}(\mathsf{skip})$ is $j' \cdot \mathsf{com} \multimap \mathsf{com}$ with $j' \oplus \mathbf{1} = \mathbf{1}$. Clearly, $j' = \mathbf{0}$ is one solution to this that works in any semiring, but that corresponds to the case in which the recursive call never happens at all, and so is not very useful; and most of the semirings we have looked at do not have other solutions.

In the call-by-name fragment of SBLL, "$j' \cdot \mathsf{com} \multimap \mathsf{com}$ with $j' \oplus \mathbf{1} = \mathbf{1}$" is the only solution to the type of $\mathsf{seq}(\mathsf{skip})$ in $\mathsf{fix}\,\lambda x.\mathsf{seq}(\mathsf{skip})(x)$, which implies that in most of the semirings we have looked at, recursion on constants can never be useful. However, SBLL is more than just its call-by-name fragment. In the flexible fragment, we find that "$j' \cdot \mathsf{com} \multimap j \cdot \mathsf{com}$ with $j' \oplus j = \mathbf{1}$" is another solution. This is very strange to someone who is used to call-by-name languages; in a call-by-name semantics, an argument can only be used while a function is running, but in this case, we have the exact opposite situation, where the argument can only be used while the function is not running.

So what sort of semantics admits a function with this sort of type? Ghica and I discussed one possibility in [22].[1] The semantics used in that paper is not explicit, but it is still basically a call-by-name semantics. However, there are a few changes. First, we disallow mixing of

[1]The paper cited presented two expansions of recursion, an expansion in space due to Ghica and the expansion in time due to me that is presented in this paragraph. The explanation of the second expansion as a semantics for functions of type $j' \cdot \mathsf{com} \multimap j \cdot \mathsf{com}$ is new for this thesis.

concurrency with recursion (par may not be used inside a recursion, including indirectly via arguments given to a higher-order recursion). This means that it is always possible to track which recursive level is executing at any given time (in a single-threaded program, only one part of the program can execute at any given time; it is possible to identify which recursive level this is on simply by counting calls via fix, and requests for arguments via fix). Next, we have a different set of internal state for each recursion level, and which we use is determined by the recursion counter. This concept was not formalized in the original paper, and will not be here; however, informally, we can see that we have a type of "$j' \cdot \mathsf{com} \multimap j \cdot \mathsf{com}$ with $j' \oplus j = \mathbf{1}$" in which j represents "times at which the recursion counter has value 0", and j' represents "times at which the recursion counter has a positive value". These are clearly disjoint, and yet just as clearly, this semantics makes it possible for a function with type $j' \cdot \mathsf{com} \multimap j \cdot \mathsf{com}$ to exist.

In order for such a semantics to be finite-state, the recursion can only admit finitely many levels of recursion (otherwise we need an infinitely large counter, and a matching infinite amount of state). However, the example that introduced this problem shows that this is unavoidable. This also gives us a bound on the amount of state required to implement bounded affine recursion: we find that when no concurrency is involved, the amount of state required to implement general recursion that is limited to k levels is thus proportional to k bits (the above semantics shows that this much is sufficient, the example introducing the problem shows that this much is necessary).

We can thus see that general recursion is not entirely at odds with the idea of a finite-state type system, so long as we have some bound on the recursion depth, and no concurrency. An obvious extension is to work out how concurrency fits into this problem, although even apparently innocuous terms like $\vdash \mathsf{fix}\, \lambda x.\mathsf{par}(x)(\mathsf{skip})$ (which runs infinitely many copies of skip in parallel) show that this is likely to be an extremely complex subject.

The construction of recursion that we have been discussing actually underscores another important point. The earlier parts of that paper, [22], implement recursion as a space unfolding: with a limited recursion depth, this is just a bounded contraction. However, the time unfolding later in that paper (and discussed in this section), which effectively contracts different recursion levels against each other, is clearly not bounded contraction because it uses an implementation where most of the term is not duplicated, and is different from sequential or synchronized contraction because those cannot handle two terms that run simultaneously (like a function and its argument in a call-by-name setting like the one we are using here). In order to implement recursion as shown above, what we needed was a type system in which it was possible to contract a function with its argument; and this ended up needing an implementation technique that was not needed elsewhere. This strongly suggests that the correct point of view is to think of "recursive contraction", contraction between a function and its argument, as yet another form of contraction that can exist within a type system. And thus, as seen earlier, which language features a finite-state language supports continues to map directly onto which forms of contraction it supports.

Part IV

USING FINITE-STATE TYPE SYSTEMS

Chapter 10

PRACTICAL FINITE-STATE LANGUAGES

Up to this point, this thesis has been focusing on the theoretical side of finite-state type systems. It is also important, however, to get a view of how these type systems can be used in practice; a mathematical type system often needs changes to be acceptable for genuine programming. For example, although Algol 60 and Idealized Algol are mathematically similar, Algol 60 is a much better language for actual practical use. This chapter discusses the practical considerations behind the use of type systems, focusing on a practical programming language Verity (which I developed during the research for this thesis) that is very similar to SCC and bSCI. (A reference implementation of this language, that compiles Verity into hardware, can be downloaded at http://veritygos.org, although at the time of writing it predates much of the research in this thesis.)

10.1 Verity syntax

When writing about type systems mathematically, there is not much need to fix all the details of the notation. For example, $\mathsf{par}(a)(\mathsf{par}(b)(c))$ and $\mathsf{seq}(\langle a, \mathsf{seq}(\langle b, c\rangle)\rangle)$ are definitely (open) bSCI terms; but what about $\mathsf{par}\, a\, \mathsf{par}\, b\, c$ or $\mathsf{seq}\langle a, \mathsf{seq}\langle b, c\rangle\rangle$ with the parentheses removed? These terms are not necessarily "wrong", just potentially ambiguous. A practical programming language needs to come to a concrete decision on what any given term means. Mathematical notations are also excessively verbose, especially given the number of seq applications in a typical practical program; in general, an infix notation like $a + b$ is more readable than the bSCI

simple-term	$::=$	$(\ \mathit{term}\ )$	M_1
	$\mid$	$\{\ \mathit{term}\ \}$	$M_1 : \mathsf{com}$
	$\mid$	$\mathit{variable}_1$	x_1
	$\mid$	$\mathit{integer}_1\ \$\ \mathit{integer}_2$	$j_1 : \mathsf{exp}_{j_2}$
	$\mid$	$\mathit{integer}_1$	j_1
	$\mid$	`skip`	skip
	$\mid$	$(\ \mathtt{-}\ \mathit{term}\ )$	$\mathsf{op}_{-}(\langle 0, M_1 \rangle)$
argument	$::=$	*simple-term*	M_1
	$\mid$	$\mathtt{!}\ \mathit{simple\text{-}term}$	$\mathsf{deref}(M_1)$
term	$::=$	$\mathit{simple\text{-}term}_1$	M_1
	$\mid$	$\mathit{term}_1\ \mathtt{:}\ \mathit{type}_2$	$M_1 : \theta_2$
	$\mid$	$\mathit{term}_1\ \mathit{argument}_2$	$M_1(M_2)$
	$\mid$	$\backslash\ \mathit{variable}_1\ \mathtt{.}\ \mathit{term}_2$	$\lambda x_1 . M_2$
	$\mid$	$\backslash\ \mathit{variable}_1\ \mathtt{:}\ \mathit{type}_2\ \mathtt{.}\ \mathit{term}_3$	$\lambda(x_1 : \theta_2) . M_3$
	$\mid$	$\backslash\ \mathtt{(}\ \mathit{variable\text{-}list}_1\ \mathtt{)}\ \mathtt{.}\ \mathit{term}_2$	(see text)
	$\mid$	$\mathit{term}_1\ \mathtt{,}\ \mathit{term}_2$	$\langle M_1, M_2 \rangle$
	$\mid$	$\mathit{term}_1\ \mathtt{;}\ \mathit{term}_2$	$\mathsf{seq}(\langle M_1, M_2 \rangle)$
	$\mid$	$\mathit{term}_1\ \mathtt{\mid\mid}\ \mathit{term}_2$	$\mathsf{par}(M_1)(M_2)$
	$\mid$	$\mathit{term}_1\ \mathit{operator}_2\ \mathit{term}_3$	$\mathsf{op}_{\bullet_2}(\langle M_1, M_2 \rangle)$
	$\mid$	$\mathtt{\tilde{}}\ \mathit{term}_1$	(see text)
	$\mid$	$\mathit{term}_1\ \mathit{constant\text{-}operator}_2\ \mathit{integer}_3$	(see text)
	$\mid$	$\mathtt{!}\ \mathit{term}_1$	$\mathsf{deref}(M_1)$
	$\mid$	$\mathit{term}_1\ \mathtt{:=}\ \mathit{term}_2$	$\mathsf{assign}(\langle M_1, M_2 \rangle)$
	$\mid$	$\mathtt{if}\ \mathit{term}_1\ \mathtt{then}\ \mathit{term}_2\ \mathtt{else}\ \mathit{term}_3$	$\mathsf{if}(\langle M_1, \langle M_2, M_3 \rangle \rangle)$
	$\mid$	$\mathtt{while}\ \mathit{term}_1\ \mathtt{do}\ \mathit{term}_2$	$\mathsf{while}(\langle M_1, M_2 \rangle)$
	$\mid$	$\mathtt{new}\ \mathit{variable}_1\ \mathtt{in}\ \mathit{term}_2$	$\mathsf{newvar}(\lambda x_1 . M_2)$
	$\mid$	$\mathtt{new}\ \mathit{variable}_1\ \mathtt{:=}\ \mathit{term}_2\ \mathtt{in}\ \mathit{term}_3$	$\mathsf{newvar}(\lambda x_1 . \mathsf{seq}(\langle \mathsf{assign}(\langle x_1, M_2 \rangle), M_3 \rangle))$
type	$::=$	`com`	exp_0
	$\mid$	$\mathtt{exp}\ \$\ \mathit{integer}_1$	exp_{j_1}
	$\mid$	`exp`	exp
	$\mid$	$\mathtt{var}\ \$\ \mathit{integer}_1$	$\mathsf{exp}_{j_1} \times (\mathsf{exp}_{j_1} \to \mathsf{com})$
	$\mid$	`var`	$\mathsf{exp} \times (\mathsf{exp} \to \mathsf{com})$
	$\mid$	$\mathit{type}_1\ \mathtt{->}\ \mathit{type}_2$	$\theta_1 \to \theta_2$
	$\mid$	$\mathit{type}_1\ \mathtt{*}\ \mathit{type}_2$	$\theta_1 \times \theta_2$
	$\mid$	$(\ \mathit{type}_1\ )$	θ_1
variable-list	$::=$	$\mathit{variable}_1$	x_1
	$\mid$	$\mathtt{(}\ \mathit{variable\text{-}list}_1\ \mathtt{)}$	M_1
	$\mid$	$\mathit{variable\text{-}list}_1\ \mathtt{,}\ \mathit{variable\text{-}list}_2$	$\langle M_1, M_2 \rangle$

Figure 10.1.1: Verity syntax corresponding to SCC

$\mathsf{op}_+(\langle a,b\rangle)$. Finally, they are difficult to input; most methods of input to practical computers fail to distinguish between, say, $\langle\rangle$ and $<>$, which can cause ambiguities in its own right. We might want both $\langle f\langle 2,3\rangle x,y\rangle$ and $\langle f < 2,3 > x,y\rangle$ to be (different) terms in a practical programming language (meaning "$\langle(f(\langle 2,3\rangle)(x)),y\rangle$" and "$\langle(\mathsf{op}_<(\langle f,2\rangle)),\langle(\mathsf{op}_>(\langle 3,x\rangle)),y\rangle\rangle$" respectively), but these notations will both be typed as `<f<2,3>x,y>` on a typical computer. Likewise, the difference between, say, if and if (the former being a constant, the latter being a variable) is one that has caused problems in practice; in both Algol 60 and Algol 68, the two are defined as being different, which has been known to cause problems distinguishing between them. As Hansen and Boom explain in [24, section 3.5], some Algol implementations will write the constant as `.if` or `IF` to disambiguate; others change the variable, accepting forms such as `_if` or `i f`. Verity's approach to the problem is one common in modern languages; `if` is always a constant (and one that requires specific sugar used around it), and variables must be given names that are unlike those used elsewhere in the syntax (thus there is no way to write if, but this is unimportant due to the infinite supply of other variable names to use).

Verity's syntax is given by a set of recursive definitions; this syntax seems to have been invented by Backus in 1959 for Algol 60 (being cited as such in [2], in addition to other papers), but I could not track down the original paper to verify this. It is commonly used to describe languages nowadays. An advantage of this format is that the syntax can be automatically verified for desirable properties by a computer. The syntax that corresponds to constructs and features that exist in SCC is shown in Figure 10.1.1, where a *variable* is a sequence of letters, digits and underscores with no other meaning and that does not start with a digit, an *integer* is a sequence of digits, and the possible *operator*s are shown in Table 10.1. The translation of this syntax to SCC is also shown, via using subscripts to show which parts of the Verity syntax correspond to which parts of the SCC. (In both of these, and in Verity generally, exp_j is defined as exp_J where $J = \{k \in \mathbb{N} \mid 0 \leq k < 2^j\}$; because most practical computers store information in binary, Verity requires the range of an expression to be contiguous, starting at 0, and with a whole number of bits.)

Notation	Meaning	Type
+	Addition	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_j$
-	Subtraction	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_j$
*	Multiplication	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_j$
<	Less than	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_1$
>	Greater than	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_1$
+<	Signed less than	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_1$
+>	Signed greater than	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_1$
==	Equal to	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_1$
|	Bitwise inclusive OR	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_j$
&	Bitwise AND	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_j$
^	Bitwise exclusive OR	$(\mathsf{exp}_j \times \mathsf{exp}_j) \to \mathsf{exp}_j$

Table 10.1: Verity binary operators

The list of operators immediately shows a difference between practical and theoretical languages; in a theoretical language, there is rarely a need to fix which operators are in use because it mostly does not matter, but for practical use, some subset needs to be chosen. Verity has some standard arithmetic operations on integers (addition, subtraction, multiplication, comparisons; division is omitted mostly because Verity aims to be implemented on hardware with no divide circuits, but partly because it is semantically awkward due to fractional results and divisions by zero). Operations on exp_j operate modulo 2^j, so that the result is always representable. However, Verity also has some operations that interpret the values in other ways (as potentially negative integers, or as a sequence of bits), allowing Verity programs more freedom in the data they manipulate. The so-called *signed* operations interpret exp_j values above or equal to 2^{j-1} as negative numbers, by adding 2^j to a negative number to get its representation (this is known as "two's complement" representation, and used by the vast majority of modern computers). This gives mostly transparent support for signed numbers, because as the signed and unsigned interpretation of a value are equal modulo 2^j, addition, multiplication, and subtraction work just as well on signed as on unsigned numbers. However, comparisons work differently; $2 : \mathsf{exp}_2$ actually means -2 when interpreted as signed, so it is signed-less than $1 : \mathsf{exp}_2$, but unsigned-greater than it. Meanwhile, the so-called *bitwise* operations work on each bit of the representation of the numbers individually; $12 : \mathsf{exp}_4$ is interpreted as $\langle \mathsf{true}, \mathsf{true}, \mathsf{false}, \mathsf{false} \rangle$, and the standard

Boolean operations (inclusive OR, exclusive OR, AND) can be used point-wise on these tuples. A degenerate case of this is exp_1, which with the bitwise view is a single Boolean, and is often used as such in Verity.

We can see from Table 10.1 that unlike in most mathematical languages, op has different types depending on which operation is used. In fact, Verity needs even more flexibility than this; there are also some operations that take only one numerical argument, rather than two (and thus, from the bSCI view, take an exp rather than an $\mathsf{exp} \times \mathsf{exp}$ as their argument). These operations are bitwise NOT, multiplication and (signed and unsigned) truncating division by 2^k for constant k, and a family of (signed and unsigned) constants of type $\mathsf{exp}_j \to \mathsf{exp}_k$ which preserve the value they are given modulo 2^k. Each has its own syntax; a prefixed `~` is used for bitwise NOT, whereas the other operations ("constant operators") are written as `<<`, `>>` and `$$` respectively, followed by the value of k, and preceded by `+` if they are to interpret their arguments and return value as signed.

The remaining unusual case is projections. In Verity, programmers do not specify projections explicitly; rather, they use a lambda with a *variable-list* as the argument. This is interpreted as a term which takes a single (tuple) argument, where every use of that argument is replaced by a projection. For example, `\(a,b).a+b` means $\lambda x.\mathsf{op}_+(\langle \pi_1(x), \pi_2(x)\rangle)$. The argument can be a nested set of tuples, in which case multiple projections are needed to parse it correctly. There is also sugar for a combined lambda and application (which is convenient for code reuse); `let x=M in N` means $(\lambda x.N)(M)$, and (for the reasons discussed in Section 8.1) an extended form `let x=M1 and y=M2 in N` which is equivalent to `(\(x,y).N)(M1,M2)` (and which then further desugars into projections, $(\lambda z.N[\pi_1(z)/x][\pi_2(z)/y])(\langle M_1, M_2\rangle)$).

One important desirable feature for a language such as this is that it is unambiguous; that is, that each sequence of characters has at most one meaning as an SCC term. With Verity as defined above, this is not the case; there are programs such as `2+3*4` or `8-4-2` which have multiple valid parses. This problem is resolved in the normal way, using precedence and associativity rules; some operations "bind tighter" than others (e.g. multiplication binds more tightly than

$$\text{tight}\quad \texttt{\$\$}\, M(N)\ \texttt{!}\ \texttt{\~{}}\ \texttt{*}\ \overbrace{\texttt{+ -}}^{\text{equal}}\ \overbrace{\texttt{<< >>}}^{\text{equal}}\ \overbrace{\texttt{< >}}^{\text{equal}}\ \texttt{== \&}\ \overbrace{\texttt{| \^{}}}^{\text{equal}}\ \texttt{:=}\ \overbrace{\texttt{while if}}^{\text{equal}}\ \texttt{;}\ \texttt{||}\ \texttt{new}\ \texttt{,}\ \overbrace{\backslash\,\texttt{let}}^{\text{equal}}\ \texttt{->}\quad \text{loose}$$

Figure 10.1.2: Precedence of Verity

addition, so the first term is `2+(3*4)`), and each ambiguous piece of syntax has an associativity rule for what happens when it is used twice in a row (e.g. with subtraction, the left subtraction is done first, making the second term `(8-4)-2`). The precedence order is shown in Figure 10.1.2; everything is left-associative apart from `,` and `->` which are right-associative. Due to the existence of program inputs like `f !a(x)` (which is otherwise ambiguous between $f(\mathsf{deref}(a))(x)$ and $f(\mathsf{deref}(a(x)))$), it is necessary to add a precedence for application in addition to the various operators, an operation that is hard to express as input to tools used to prove unambiguity of grammars (because application does not necessarily use any specific sequence of characters that can be recognized). A consequence is that although a version of Verity's grammar has been proved unambiguous, that version is one with sufficient extra complexity over the version in Figure 10.1.1 that there would not be much benefit to reproducing the proof here.

10.2 Practical type systems

Verity is based on SCC, so its type system might be expected to be identical to that of SCC. However, it is common to have multiple possible representations of a type system, each of which admit the same terms. Likewise, although two equivalent programs have equal denotational semantics (by definition), the way in which those semantics are produced can have a practical difference. As an example, consider a categorical semantics which contains morphisms $\delta_{[\![\theta]\!]} : [\![\theta]\!] \longrightarrow [\![\theta]\!] \times [\![\theta]\!]$ and $[\![x : \theta \times \theta \vdash \pi_1(x) : \theta]\!] : I \otimes (A \times A) \longrightarrow A$ (which would be expected in a semantics of SCI, which has products rather than tensors). $(\mathbf{id}_I \otimes \delta_{[\![\theta]\!]}); [\![x : \theta \times \theta \vdash \pi_1(x) : \theta]\!]; \rho^{-1}_{[\![\theta]\!]}; \gamma_{I,[\![\theta]\!]}$ would be a valid morphism in this semantics, and would most likely be equal to $\mathbf{id}_{I\otimes[\![\theta]\!]}$, inside the category. But although these two denotations are semantically equal (i.e. they have the same meaning, and do the same thing), in a practical language

$$\dfrac{\dfrac{\dfrac{}{x:\theta_1 \vdash x:\theta_1}\text{Identity}}{x:\theta_1, y:\theta_2 \vdash x:\theta_1}\text{Weakening} \qquad \dfrac{\dfrac{}{y:\theta_2 \vdash y:\theta_2}\text{Identity}}{x:\theta_1, y:\theta_2 \vdash y:\theta_2}\text{Weakening}}{x:\theta_1, y:\theta_2 \vdash \langle x,y\rangle : \theta_1 \times \theta_2}\text{Product}$$

Figure 10.2.1: Deriving $\langle x,y\rangle$ with implicit contraction requires Weakening

implementation that worked inductively on the type derivation, such as the one used in the reference implementation, the implementations would be different (the first would be produced via actually combining a diagonal and a projection, possibly with extra bookkeeping; and implementations would not necessarily have to be aware that they canceled each other out). In particular, the simpler implementation would be more efficient. One way to look at the situation is that the denotational semantics describe the behaviour of the implementation, but they are not themselves the implementation, and two different implementations of a term can (in fact, must) thus have the same semantics, just as two different terms can have the same semantics.

Thus, it makes sense to write the type system in a way that aims to keep the resulting implementations simple. In particular, we want to avoid the situation seen in many type systems where the derivation of $\langle x,y\rangle$ requires Weakening to be used on both x and y. (This happens in, say, the type system of Figure 3.3.2; the derivation looks like Figure 10.2.1, and Weakening is required because as both premises of the Product rule need identical contexts, they must both mention both x and y in the context despite mentioning only one in the term itself, and Weakening is the only way to adjust contexts like this). Forming a diagonal only to discard one of the copies is significant extra complexity. The main practical implication of this is that we do not want to use the implicitly-contracting version of the Product rule, as seen for bSCI in Figure 8.1.1. However, an explicitly contracting version of the type system has its own problems; as explained in Section 8.2, explicit contraction in SCI-like languages requires existential typing, greatly complicating the language. The correct version is somewhere in between; we want implicit contraction when we need it, but not otherwise. To accomplish this, we remove the explicit Weakening rule, and instead make it implicit in Product (by not copying), Abstraction (via adding a second Abstraction rule for an unused variable), and Identity and Contraction

$$\theta ::= \mathsf{exp}_j \mid j \cdot \theta \to \theta \mid \theta \times \theta, \text{ where } j \in \mathbb{N} \text{ is a nonnegative integer}$$
$$\Gamma ::= \text{set of } x \colon j \cdot \theta, \text{ where } j \in \mathbb{N} \text{ is a nonnegative integer}$$
$$\mathsf{com} \triangleq \mathsf{exp}_0$$

$$\frac{x \neq y}{x \colon \theta \# y \colon \theta'} \qquad \frac{\Gamma \# \Delta \qquad \Gamma' \# \Delta}{\Gamma :: \Gamma' \# \Delta} \qquad \frac{x \colon \theta \# \Gamma}{x \# \Gamma}$$

$$\frac{}{\mathsf{exp}_J \leq \mathsf{exp}_J} \qquad \frac{j_1 \leq j_2 \qquad \theta_2' \leq \theta_1' \qquad \theta_1 \leq \theta_2}{j_1 \cdot \theta_1' \to \theta_1 \leq j_2 \cdot \theta_2' \to \theta_2} \qquad \frac{\theta_1 \leq \theta_2 \qquad \theta_1' \leq \theta_2'}{\theta_1 \times \theta_1' \leq \theta_2 \times \theta_2'}$$

$$\frac{j \geq 1}{x \colon j \cdot \theta \vdash x \colon \theta}\ \text{Identity}$$

$$\frac{\Gamma \vdash M \colon \theta \qquad \theta \leq \theta'}{\Gamma \vdash M \colon \theta'}\ \text{Subtyping}$$

$$\frac{\Gamma \vdash M \colon \theta' \qquad \theta \text{ and } \theta' \text{ are equal up to bounds, where } \mathsf{exp} \text{ equals any } \mathsf{exp}_j}{\Gamma \vdash (M \colon \theta) \colon \theta'}\ \text{Annotation}$$

$$\frac{M \colon \theta \text{ is a constant}}{\emptyset \vdash M \colon \theta}\ \text{Constant}$$

$$\frac{x \colon 1 \cdot \theta \vdash M \colon \theta}{\emptyset \vdash \mathsf{fix}\, \lambda x.M \colon \theta}\ \text{Recursion}$$

$$\frac{\Gamma \cup \{x \colon j \cdot \theta'\} \vdash M \colon \theta \qquad x \# \Gamma}{\Gamma \vdash \lambda(x \colon j \cdot \theta').M \colon (j \cdot \theta' \to \theta)}\ I\text{-Abstraction}$$

$$\frac{\Gamma \vdash M \colon \theta \qquad x \# \Gamma}{\Gamma \vdash \lambda(x \colon j \cdot \theta').M \colon (j \cdot \theta' \to \theta)}\ K\text{-Abstraction}$$

$$\frac{\Gamma \vdash M \colon (j \cdot \theta' \to \theta) \qquad \Delta \vdash N \colon \theta' \qquad \Gamma \# \Delta}{\Gamma \cup j \cdot \Delta \vdash MN \colon \theta}\ \text{Application}$$

$$\frac{\Gamma_M \cup \Delta \vdash M \colon \theta \qquad \Gamma_N \cup \Delta \vdash N \colon \theta' \qquad \Gamma_M \# \Delta \qquad \Gamma_N \# \Delta \qquad \Gamma_M \# \Gamma_N}{\Gamma_M \cup \Gamma_N \cup \Delta \vdash \langle M, N \rangle \colon (\theta \times \theta')}\ \text{Product}$$

$$\frac{\Gamma \cup \{x \colon j \cdot \theta', y \colon j' \cdot \theta'\} \vdash M \colon \theta \qquad k \geq j + j'}{\Gamma \cup \{x \colon k \cdot \theta'\} \vdash M[x/y] \colon \theta}\ \text{Contraction}$$

Figure 10.2.2: Verity’s type system

(via allowing bounds higher than strictly necessary), in the same style as in Kfoury's $\boldsymbol{\lambda}$ (Figure 6.4.1).

The type system used by Verity is shown in Figure 10.2.2; as can be seen, it is SCC's type system (plus the affine recursion rule from Chapter 9) given in a notation where the changes suggested above are made, sets rather than sequences are used for contexts (this removes the need for the Exchange rule and so simplifies the resulting implementations), and one other change is made: the addition of an Annotation rule that allows types to be embedded in the syntax of the term itself. This is reflected in the Verity syntax in Figure 10.1.1, which allows but does not require the type of a term to be specified. The idea here is that, for debugging purposes or for disambiguation when there is more than one possible type, the user might want to specify the type for a term explicitly, or might want to leave it to be inferred. We assume that the user never wants to specify concurrency bounds (which are "below the level" at which a programmer usually thinks), and might or might not want to specify the bounds of an exp.

Just as with Bounded ICA (Corollary 6.2.11), Verity's type system has decidable inference; the proof proceeds along almost the same lines. (The main change required to convert the decidability proof for Bounded ICA into a decidability proof for Verity is to Figure 6.2.3; the expanded derivations for Identity and Contraction are now more direct, and make use of the fact that they can derive terms with a higher contraction bound on the relevant variable than would seem to be necessary.) Several changes to the inference algorithm used in that proof are used in the reference Verity implementation to make it run within a reasonable length of time and produce more "minimal" results. We only consider one possible shape for the proof tree (that in which all Contractions happen immediately after an Application and only contract variables coming from one side of the application with variables from the other side). This excludes some implementations that would otherwise be allowed (such as that in Figure 10.2.3a); however, in all cases I know of, there is some derivation of the term with a different implementation (such as that in Figure 10.2.3b; note that this is using sequential contraction rather than bounded contraction, and as such is a materially different implementation). Similarly, only one choice of

$$\dfrac{\dfrac{}{\emptyset \vdash \mathsf{seq} : 1\cdot(\mathsf{com}\times\mathsf{com})\to\mathsf{com}} \quad \dfrac{\dfrac{\dfrac{}{\{x:1\cdot\mathsf{com}\}\vdash x:\mathsf{com}} \quad \dfrac{}{\{y:1\cdot\mathsf{com}\}\vdash y:\mathsf{com}}}{\{x:1\cdot\mathsf{com}, y:1\cdot\mathsf{com}\}\vdash \langle x,y\rangle : (\mathsf{com}\times\mathsf{com})}}{\{x:2\cdot\mathsf{com}\}\vdash\langle x,x\rangle:(\mathsf{com}\times\mathsf{com})}}{\{x:2\cdot\mathsf{com}\}\vdash \mathsf{seq}(\langle x,x\rangle):\mathsf{com}}$$

(a) Well-typed Verity, but cannot be inferred

$$\dfrac{\dfrac{}{\emptyset \vdash \mathsf{seq} : 1\cdot(\mathsf{com}\times\mathsf{com})\to\mathsf{com}} \quad \dfrac{\dfrac{}{\{x:1\cdot\mathsf{com}\}\vdash x:\mathsf{com}} \quad \dfrac{}{\{x:1\cdot\mathsf{com}\}\vdash x:\mathsf{com}}}{\{x:1\cdot\mathsf{com}\}\vdash\langle x,x\rangle:(\mathsf{com}\times\mathsf{com})}}{\{x:1\cdot\mathsf{com}\}\vdash \mathsf{seq}(\langle x,x\rangle):\mathsf{com}}$$

(b) An inferable derivation of the same term

Figure 10.2.3: The effects of simplified type inference

which bounds are 0, which bounds are 1, and which bounds are more than 1 is used; all bounds that can possibly be 0 are set to 0, then all bounds that can possibly be 1 are set to 1, leaving the others as more than 1.

These optimizations are obviously sound; if they find a derivation, it will be correct. An open problem is as to whether they are complete; it is unknown whether there are any terms that have types, but for which the optimized algorithm cannot find any types.

10.3 Separate compilation

Defining a denotational semantics via structural induction, as seen in Section 5.2, has one large advantage: such semantics are *compositional*. This means that from the denotation of M and the denotation of N, you can calculate the denotation of $M(N)$ even without knowing what M and N are. This is similar to a practically useful property that most languages need to be usable for more than small projects: you can produce implementations for M and N separately ("compiling" M and N), and then a separate "linking" step allows you to produce an implementation of $M(N)$ without needing to reproduce the implementations of M and N.

There are three main reasons why you would want to do this. One is to save time in compiling; it means that changing a small part of a large program does not require the entire program to be recompiled, which can greatly speed up a workflow. Another is that it allows an implementation of a program to be distributed to other developers as a mostly opaque object, meaning that it can be distributed even in formats that cannot meaningfully be manipulated (e.g. in the case of the reference Verity implementation, which compiles into hardware, it would be possible to distribute an implementation of a program as a piece of physical hardware, which is generally almost impossible to alter once it has been created). The third reason, and possibly the most important, is that nothing about separate compilation precludes the linking of an implementation compiled from one language with an implementation compiled from a different language (so long as there is some category which has, for each language, a denotational semantics of that language as a subcategory); this is known as a "foreign function interface" or FFI. This is frequently used in practice to implement parts of programs that would be inappropriate or difficult to write in the same language as the rest of the program. For example, modern computers run a range of different operating systems, with a range of different methods for displaying text to the user. If a programming language implementation wants to provide a method of displaying text to the user, one method would be to have different code for each operating system, but such a method would be hard to maintain, as the code would need updating each time a new operating system was produced. Instead, the most common approach is to use an FFI to the C programming language (which is very widely implemented), which centralizes the effort of maintaining code for operating-system-specific functions in one place (the C implementation).

The practical problem involved here is that although producing $[\![M(N)]\!]$ from $[\![M]\!]$ and $[\![N]\!]$ is trivial in any compositional implementation (this is the Application rule), practical uses of separate compilation or an FFI do not generally look like this, mathematically. Programmers generally do not think in terms of applying one program to another; rather, they think in terms of "libraries" which contain code that can be used by their main program, or other libraries. In Verity, although libraries (like everything else) are effectively a single SCI term, programmers think of them as providing one or more functions. Thus, there is something of a mismatch between the way in which an FFI works mathematically, and the way in which programmers want to use it.

arithmetic.ia

```
let add = \(x, y). (x:exp$32) + y in
let subtract = \(x, y). add ((-x), y)
and add = add in
let multiply = \(x, y). (x:exp$32) * y in
export (add, subtract) multiply
```

main.ia

```
import "arithmetic"
import <print>
new x := add (10, 10) in
print (subtract (!x, multiply (3, 4)))
```

The linker generates:

```
(\export.print)(\print.(\export.arithmetic)
                          (\(add, subtract).\multiply.main))
```

Figure 10.3.1: Separate compilation example in Verity

$$\begin{aligned}\mathbf{f}(export, N) &\triangleq N \\ \mathbf{f}(M'(M''), N) &\triangleq \mathbf{f}(M', \lambda M''.N)\end{aligned}$$

Figure 10.3.2: Conversion of `export` to lambdas

The solution used in Verity is for the linker to generate a separate file that bridges the gap between these two models; a programmer writes the program as if it is using functions from another file, and the linker will produce a wrapper that makes everything into a single function. An example is shown in Figure 10.3.1, using a Verity approximation of the wrapper generated by the linker (the linker actually generates an implementation directly, for reasons explained below).

The basic principle is quite simple. Linking is done at the bSCI level, after bounded contraction has already been eliminated. Libraries and programs are both implemented as open terms; libraries have a free variable `export`, and anything that imports functions from a library has free variables to represent those functions. An extra language construct, `import`, is added to allow a program to specify which libraries it depends on (`import "library"` for a custom library, or `import <library>` for a library built in to the Verity compilation system); this has no effect on the actual term represented by the program, but is used during type inference (so that the free variables in the program will have the correct types), and during linking (to work out which compiled programs need to be linked in, and what functions they export). An examination of the source is used to determine the names of the functions being exported (which are just arguments to `export` in the original program), and lambdas are added to the program doing the import to match the structure of the export. (The export needs to exist in both curried and uncurried forms due to the way in which SCI indicates sharing, and the structure is not present in the main program.)

Here is how the process is formalized. Given a program N and a library M, the two can be combined into the term $(\lambda export.M)(\mathbf{f}(M',N))$, where "**f**" is defined in Figure 10.3.2 and M' is the `export` construct in M. Repeating this process recursively allows any number of libraries to be combined with a program. It is also possible for libraries to import from each other, so long as the dependencies are acyclic and no exported function or tuple of exported functions is imported more than once; the dependencies must be acyclic because an importing library must come after the exporting library, and no function or tuple can be imported more than once

$$\frac{\Gamma \cup \{x\colon\theta_1', y\colon\theta_2'\} \vdash M\colon\theta \qquad \Gamma \cup \{z\colon\theta_1' \times \theta_2'\} \vdash M[\pi_1 z/x][\pi_2/y]\colon\theta \qquad x\#\Gamma,\, y\#\Gamma,\, z\#\Gamma}{\Gamma \cup \{z\colon\theta_1' \times \theta_2'\} \vdash (\lambda x.\lambda y.M(\pi_1 z))(\pi_2 z)\colon\theta}\ \text{Link}$$

Figure 10.3.3: Expanding Verity to handle linking

because otherwise the resulting code will not type in SCC as it will try to share a free variable across an application. This is implemented simply by importing the combining the libraries with the program one at a time: the libraries are topologically sorted into a dependency order, then a most-depending library is combined with the program, then the other libraries in order, until finally a most-depended library is combined with the combination of the program and all the other libraries.

There is one major subtlety, though. The above process does not directly handle tuples; if given an input that contains tuples, it can produce terms such as $\lambda\langle x,y\rangle.N$, which is not syntactically valid SCC (or even ICA). We would ideally want to interpret this as $\lambda z.N[\pi_1 z/x][\pi_2 z/y]$ (which is how Verity interprets `\(x,y).N`); this types correctly, but is no longer compositional (as knowledge of the internals of N is needed to do the substitutions correctly). An obvious alternative is $\lambda z.(((\lambda x.\lambda y.N)(\pi_1 z))(\pi_2 z))$, which is correctly compositional and obviously semantically equivalent to the above term; however, this term is no longer valid bSCI, because z appears free on both sides of an application (in yet another example of why it causes problems for beta-reduction to not be an equivalence).

As far as I know, there is no solution to this problem in bSCI itself (although the latter alternative works in TCI). The Verity reference implementation uses a different solution: it effectively adds a new rule to the type system (with a matching denotation), shown in Figure 10.3.3. This rule is unusual in that it only applies when two different variations on the same term both type; the denotation of $(\lambda x.\lambda y.M(\pi_1 z))(\pi_2 z)$ is defined entirely in terms of $\Gamma \cup \{x\colon\theta_1', y\colon\theta_2'\} \vdash M\colon\theta$, but the derivation is only valid if $\Gamma \cup \{z\colon\theta_1' \times \theta_2'\} \vdash M[\pi_1 z/x][\pi_2 z/y]\colon\theta$ also happens to type. The reference implementation will, when compiling a term M, check whether $M[\pi_1 z/x][\pi_2 z/y]$ would type correctly for every choice of free x and free y in the program; it records this information along with the implementation of the term. This then allows the linker to know whether the Link rule is applicable or not without needing to inspect M directly; it can use the recorded information without needing to know anything about M. Other information is also recorded to allow the correct functioning of the linker; for example, the context Γ and type θ must be recorded.

Although I produced this linking method for the Verity reference implementation, the new typing rule needed to allow linking to work in SCI has some similarities to bunched typing (introduced by O'Hearn and Pym in [36]). In particular, "$\Gamma, x, y \vdash M : \theta$ where $\Gamma, z \vdash M[\pi_1 z/x][\pi_2 z/y]$ would type correctly" behaves somewhat like a judgement in its own right; in the $\alpha\lambda$-calculus (explained by O'Hearn in [34]), the judgement can be written as $\Gamma, (x; y) \vdash M : \theta$. The Link rule given above is hard to prove results about, due to its unusual structure; it seems possible that expressing separate compilation in terms of bunched typing would work better. Likewise, it is only used in the linker (which is why the link wrapper has to be produced directly, rather than going via Verity; the type-checker used for other Verity code would reject it).

There are some other caveats associated with separate compilation. Most notable is that it places limits on type inference; if everything were in one program, the types inferred for the arguments to a library function would depend on the types with which they were used in the program that imports it, but because the library might be completely compiled before the program is even written, this can no longer be the case. This explains the reason why explicit `:exp$32` annotations were needed in Figure 10.3.1; `<print>` exports $\mathit{print} : (\mathsf{exp}_{32} \to \mathsf{com})$, but `"arithmetic"` cannot use this information because it does not know it will be linked against `<print>`. For related reasons, all exported functions have contraction bounds of 1; the extra power granted by SCC is useful only within one program or library, with the bridge between them effectively being just SCI.

Chapter 11

CONCLUSIONS

In this thesis, we have looked at many different type systems within the spectrum of languages that is effectively bounded below by Affine ICA, which has no contraction, and above by ICA, which allows unlimited contraction (and thus is not finite state). In many papers on programming languages, the focus is on what the type systems allow and reject. Although important, this can miss the important consideration of efficiency; for example, although all ISBLL terms (and thus all TCI terms) type in Intersecting ICA (Theorem 7.4.1), synchronized and sequential contraction are often going to be more efficient implementations than bounded contraction (which can require duplicating large amounts of code). Thus, we have focused not just on what is possible while remaining finite-state, but also on what is possible given specific implementations of finite-state contraction.

11.1 Language features as contraction rules

Although the languages we have looked at have sometimes looked quite different from each other, there are underlying patterns. One of the main goals of this thesis is to demonstrate that in the finite-state setting, language features normally correspond to contraction rules. We have seen how bounded contraction can be implemented as a syntactic transformation via linearization, meaning that almost any implementation can gain the power of effectively unrestricted finite-state contraction if it is willing to pay the cost of duplicating code. We have seen how pipelining, which is often used as an optimization when using implementations that are naturally highly concurrent (such as hardware), is a natural consequence of a different sort of contraction rule, synchronized contraction. Interestingly, the same concept of synchronized contraction captures the essence of both the synchronous pipelines that are most commonly found in hardware research (which are similar to the hard real-time systems discussed in Section 7.2), and the asynchronous pipelines that were famously popularized by Sutherland in [46] (which are similar to the local-timescale systems discussed in Section 7.4), thus providing striking evidence that language features can often be viewed through the lens of contraction even if they originated in substantially different circumstances. We have also seen how the SCI family of languages uses sequential contraction, and that although the language looks quite unfamiliar when written with an explicit contraction rule, most of the differences are either due to the fact that SCI was not originally designed as a finite-state language, or due to the existentially typed nature of its product. Even general recursion turned out to be perhaps best viewed as a form of contraction.

It is interesting to look back at one of the relevant papers with this view in mind. O'Hearn's paper [34] contained a section on the motivation for bunched typing, and particularly relevant is its subsection 2.1, which discusses the conceptual difference between SCI and linear logic. As O'Hearn explains, linear logic is based around controlling consumption and duplication of data, whereas SCI is based around controlling sharing of data. One of the motivations for the paper in question was to capture whether the difference was a matter of different semantics for the same type system, or whether the two type systems themselves were necessarily separate, and the paper comes to the latter conclusion (Proposition 15 on page 26 of that paper shows that no implementation of linear logic's ! functor can be used to implement all models of $\alpha\lambda$-calculus, the generalization of SCI that that paper introduces).

From the point of view of this thesis, we can get another view of this result. Linear logic's ! naturally involves infinities, but we can instead think in terms of the $!_j$ from bounded linear logic (which in turn is SBLL using the integers as the semiring in question, although bounded linear logic historically came first; this thesis calls the operation in question $j\cdot$). We can thus think of Bounded ICA as a subset of linear logic (if one that adds many extra restrictions); O'Hearn's view of linear logic is what this thesis calls bounded contraction. Meanwhile, as discussed earlier, SCI is about a rather different sort of contraction, sequential contraction. O'Hearn's result thus effectively discovered that sequential and bounded contraction are different concepts, even if they look quite similar at first. In the notation of this thesis, it says that given a type system with two sorts of function arrows, $\rightarrow$ that allows contraction between the function and argument and $\multimap$ that does not, then if the type system's only other form of contraction (apart from contraction around $\rightarrow$) is bounded contraction, $\theta' \rightarrow \theta$ can always be replaced with some $j \cdot \theta' \multimap \theta$ while maintaining typing (given the correct choice for the semiring from which j is drawn; O'Hearn's choice allowed infinite j), but if its only other form of contraction is sequential contraction, then for some selections of constants, the same replacement is not always possible. Or in more informal language: there are terms that bounded contraction can type and sequential contraction can't. When expressed like this, it becomes clear why SCI and linear logic are different.

Viewing language features as contraction rules also gives a roadmap to combining them. SCC allows both bounded and sequential contraction, but the way that SCC is defined makes the interaction between them unclear (and as is explained in Section 7.3, this has historically lead to inconsistencies in the definition of SCC). Meanwhile, adding bounded contraction to TCI (to produce an analogue to SCC, the same way that TCI is analogous to SCI) has only one obvious implementation: remove the $\theta \# \theta'$ side conditions on contraction and intersection (and then implement contractions and intersections where $\theta \# \theta'$ happens to hold using sequential contraction, and other contractions and intersections using bounded contraction). Likewise, if we wanted a language to describe an implementation that supported both synchronized and sequential contraction, we could just use ISBLL, using synchronized contraction where necessary (i.e. where an element υ of $\mathbb{A}$ is used for more than one purpose), and sequential contraction elsewhere.

Interestingly, the various contraction methods seem to form a hierarchy. Bounded contraction can be used to emulate synchronized contraction, but less efficiently (it requires duplication of code, and higher-order constants need to be able to handle arbitrary bounds on the arguments to their arguments). Likewise, synchronized contraction can be used to emulate sequential contraction, again less efficiently (requiring the use of potentially contended locks, which complicate an implementation and slow down the program). Even recursive contraction seems like it might fit in, at the top of the hierarchy; if you have a contraction method that can contract a function with its argument, you can just use it everywhere, and there will be no need for copying. (However, I am not completely sure that this equivalence holds, because it does not obviously handle tensors; the $(\theta'' \otimes \theta') \to \theta \equiv \theta'' \to (\theta' \to \theta)$ equivalence suggests that there is some way to make it work, but I have not yet found such a method.) It is unclear to me whether this hierarchy is something inherent in the nature of contraction, or just a coincidence. I currently suspect the latter (and thus that there are two language features, with corresponding forms of contraction, for which neither can be implemented in terms of the other), but a result either way would be interesting, either discovering new language features of interest in a finite-state setting, or else (in effect) showing that arbitrary Intersecting ICA programs can be implemented in an optimized way via replacing more expensive with less expensive forms of contraction. (And if, as I suspect, all ICA programs that do not use semaphores or fix type in intersecting ICA, this result would generalize to ICA as a whole.)

11.2 Pitfalls in type system design

In addition to the pattern discussed in the previous section, another pattern has become apparent in finite-state type systems: a surprisingly large proportion of such type systems, both pre-existing ones and ones I designed for this thesis, have had design flaws that caused them to be unable to type all terms that could be implemented via the language features implied by their contraction rules. This is both practically relevant, because some programs that look correct will fail to type in a practical implementation; and mathematically relevant because it implies that a semantics designed based on the language features that the language was intended to model could not be fully abstract (i.e. there would be some denotations within the semantics which were not the denotation of any term of the language). Examples of languages where this has happened include bSCI, SCC, Bounded ICA, call-by-name SBLL, and flexible SBLL, in addition to several other languages that I designed and then rejected in the course of designing this thesis (such as an attempt to make a language for sequential contraction as an instance of SBLL, and the call-by-name fragment of TCI, which was present in this thesis for a long time before I realized that a larger fragment was necessary for tensors to work correctly).

The results in this thesis go some direction towards explaining this pattern. Probably the largest result in this direction is Theorem 6.3.2, which effectively says that when designing a language to describe the behaviour of programs, that language's contraction or application behaviour cannot be a subset of the contraction or application behaviour of simply typed lambda calculus. This gives a good guide as to what was going wrong: the contraction and application rules of simply typed lambda calculus are very familiar, and when trying to create a "smaller" language, it is natural to attempt to do so simply by placing side conditions on them. If the reason we want to create a smaller language is so that properties of the terms are described by their types, this will not work. The essence of the problem is that terms that are monomorphic in lambda calculus can become polymorphic once the type system becomes more precise. There are several potential solutions, such as polymorphism, intersection typing, or dependent typing.

A smaller mistake, but one that I nonetheless made in a number of languages over the course of developing this thesis, is to stick entirely to call-by-name fragments of languages. As Section 7.3 explains, this severely restricts what can be done with tensors: either any information tracked in the tensors has to treat both halves of the tensor identically, or else tensors can only appear on the left hand side of a function arrow. This leads to practical problems too, most obviously in the SCC-based Verity: if a function takes two arguments via a product (which is necessary to be able to perform sequential contraction on them), both of those arguments must have the same contraction bounds. Thus, if one of those arguments is used several times and the other argument only once, that argument will need to be duplicated several times, and most of the copies thrown away. Unlike the polymorphism problem (which also affects Verity), this does not outright prevent the program from compiling; but the result is rather less efficient than it could be. Restricting tensors to only appear on the left hand side of a function arrow is not really an option for fixing this in the case of Verity, as this makes it impossible for functions to return variables, an operation that frequently occurs in existing Verity code. The correct solution is most likely to add a little extra flexibility into the type system, via allowing contraction bounds both inside and outside the tensors.

For the sake of completeness, it is probably worth also mentioning the need to distinguish between tensors and products, using projections with products and uncurry for tensors, as explained in Section 3.2. This is not a new result, but rather a lesson that has been learned many times, including before the work on this thesis started. However, it is a mistake that still seems to be made often enough that it deserves mentioning in a section on pitfalls in type systems.

Finally, this thesis goes some way towards explaining why models of sequential contraction are so hard to find; for example, TCI took me well over a year to develop. As explained in Section 8.2, the reason is that sequential contraction is a form of existential typing; attempts to fit it into a non-existentially-typed framework like SBLL are thus unlikely to work. This likely also explains the difficulties that have historically existed in finding a version of SCI in which beta reduction is an equivalence; the effective $\exists\tau.$ qualifiers that are implied by the products in SCI make it hard to abstract out portions of code, because if the quantifiers are implied by some other language feature (and thus are not explicit), apparently innocuous operations like beta reduction can leave nowhere to put the quantifier. Hopefully, this view of sequential contraction will go some way towards explaining the other peculiarities of SCI-based languages, too.

11.3 Future directions

This thesis has been focused on finite-state type systems, identifying common patterns and common pitfalls. Although it has made major progress in this direction, it has raised a lot of new questions in the process.

One obvious direction for improvement is that of type inference. The problem is, given a term, to discover what types it has. This thesis gives an algorithm for doing that for Bounded ICA, but that algorithm is too inefficient to be practically useful for anything but tiny programs.

The reference Verity compiler has its own variant of that algorithm (for SCC), which is much more efficient, but which has not been proved correct. Finding an efficient and correct version of that algorithm would be practically useful in the short term.

However, as seen earlier, Bounded ICA and SCC are not ideal languages to be using for practical finite-state work anyway. Type inference algorithms for Intersecting ICA, ISBLL, and TCI would all be practically useful, probably more so than inference algorithms for Bounded ICA and SCC (and would also most likely incidentally answer the currently open question of whether all ICA terms that do not use fix or semaphores type in Intersecting ICA). Because it seems highly likely to produce practically useful results, this will be one of my main focuses for my research after this thesis.

A different tack would be to consider the details of implementations of these languages. I was originally planning to write this thesis about implementations of SCI-based languages in terms of asynchronous hardware. However, I had to abandon that plan because none of the existing formalizations of asynchronous hardware were appropriate for this sort of research. An existing model by Snepscheut in [45] gives a trace model for delay-insensitive asynchronous hardware that is strikingly similar to that of game semantics, a particular sort of denotational semantics that can also be used as a guide to implementation in the same way as an operational semantics. This basic correspondence is the focus of the "Geometry of Synthesis" series by Ghica; the original paper [15] does not mention either explicitly, but the correspondence becomes clearer in a follow-up technical report [16] and paper [19]. The problem is that the trace models have not been proved to form a category in the way that this construction expects. There has been some research in this direction: Ebergen introduced a technique [11, 12] for ensuring that the trace models were finite-state, and Udding showed that delay-insensitive circuits formed a category in [47]. However, Udding's category disallows some circuits that are required by Ghica's construction; in particular, it cannot handle the Weakening rule correctly (in addition to a few other constructs, such as infinite loops). Udding suggested expanding the category to handle such circuits as a topic for future development, but as far as I can tell, this never happened.

Some time after dropping the hardware content from this thesis due to discovering that not enough was known about asynchronous hardware, I also discovered that not enough was known about SCI, either. Thus, I wrote the present thesis in order to produce enough of a mathematical background on the type systems that implementing them well would be possible. It would definitely be a practically useful field of research to bridge the gap between the current knowledge of asynchronous circuits and game semantics. In particular, I strongly suspect that with an appropriate formalization of circuits, game semantics without justification pointers is a special case of asynchronous circuits. This would allow for much simpler proofs than the proof I presented in [19].

Tying into this, another useful direction would be to produce a concrete semantics – especially a game semantics – for the new languages introduced in this section. A game semantics of ISBLL or TCI in particular would be highly useful as a basis for practical compilation, because it would allow expanding the existing Verity compiler to handle more programs and to be more efficient. I have already developed a game semantics for Affine ICA that does not use justification pointers, to use as a starting point for this; this semantics was removed from the thesis for space and relevance reasons. Expanding this to handle various sorts of contraction would be a worthy goal.

Finally, full abstraction results would be useful. In particular, producing a semantics based on language features, and then working backwards to a type system from there, would serve as a means of checking that the type system captured the nature of that form of contraction exactly, rather than being too large or too small. This is not likely to be an easy task.

List of References

[1] S. Abramsky and G. McCusker. Linearity, Sharing and State: a fully abstract game semantics for Idealized Algol with active expressions: Extended Abstract. *Electronic Notes in Theoretical Computer Science*, 3:2–14, 1996. Linear Logic 96 Tokyo Meeting. 29

[2] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, A. Woodger, R. M. De Morgan, I. D. Hill, and B. A. Wichmann. Modified Report on the Algorithmic Language ALGOL 60. *The Computer Journal*, 19(4):364–379, 1976. 25, 159

[3] S. Brookes. The Essence of Parallel Algol. Technical Report CMU-CS-97-124, Carnegie Mellon University, April 1997. `http://reports-archive.adm.cs.cmu.edu/anon/1997/CMU-CS-97-124.ps`. 28

[4] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A Core Quantitative Coeffect Calculus. In Z. Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer Berlin Heidelberg, 2014. `http://old-lipn.univ-paris13.fr/~mazza/papers/CoreQuantCoeff.pdf`. vii, 107, 109, 123

[5] B. Buyukkurt, Z. Guo, and W. Najjar. Impact of Loop Unrolling on Area, Throughput and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs. In K. Bertels, J. Cardoso, and S. Vassiliadis, editors, *Reconfigurable Computing: Architectures and Applications*, volume 3985 of *Lecture Notes in Computer Science*, pages 401–412. Springer Berlin / Heidelberg, 2006. 10.1007/11802839_48. 7

[6] Celoxica. Handel-C Reference Manual. Retrieved May 2011 via `http://www.celoxica.com`. 7

[7] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936. 16

[8] A. Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. 20

[9] M. Coppo and M. Dezani-Ciancaglini. A New Type Assignment for λ-Terms. *Archiv für mathematisch Logik und Grundlagenforschung*, 19:139–156, 1978. `http://www.digizeitschriften.de/dms/img/?PPN=PPN379931524_0019&DMDID=dmdlog16`. 91

[10] M. Davis. Hilbert's Tenth Problem is Unsolvable. *The American Mathematical Monthly*, 80(3):233–269, 1973. 77

[11] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. PhD thesis, Eindhoven University of Technology, 1987. `http://alexandria.tue.nl/extra3/proefschrift/PRF5B/8708958.pdf`. 178

[12] J. C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. Technical Report 88/10, Department of Mathematics and Computing Science of Eindhoven University of Technology, May 1988. `http://alexandria.tue.nl/extra1/wskrap/publichtml/198810.pdf`. 178

[13] G. Ferizis. *Mapping Recursive Functions To Reconfigurable Hardware*. PhD thesis, The University of New South Wales, 2005. `http://unsworks.unsw.edu.au/fapi/datastream/unsworks:859/SOURCE02`. 57

[14] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 357–370. ACM, 2013. `http://www.cis.upenn.edu/~bcpierce/papers/sized-types-for-DP.popl12.pdf`. 6, 91

[15] D. R. Ghica. Geometry of Synthesis: A structured approach to VLSI design. In *Conference Record of POPL 2007: The 34th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages*, pages 363–375, 2007. `http://www.cs.bham.ac.uk/~drg/papers/popl07x.pdf`. 7, 178

[16] D. R. Ghica. Function Interface Models for Hardware Compilation: Types, Signatures and Protocols. Technical Report CSR-08-04, University of Birmingham, 2009. `http://arxiv.org/abs/0907.0749v1`. 178

[17] D. R. Ghica and A. S. Murawski. Angelic Semantics of Fine-Grained Concurrency. *Annals of Pure and Applied Logic*, 151(2–3):89–114, 2008. First Games for Logic and Programming Languages Workshop. `http://www.cs.bham.ac.uk/~drg/papers/apal2008.pdf`. 6, 27

[18] D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic Control of Concurrency. *Theoretical Computer Science*, 350(2–3):234–251, 2006. Automata, Languages and Programming: Logic and Semantics (ICALP-B 2004). `http://www.cs.bham.ac.uk/~drg/papers/scc-tcs.pdf`. 5, 29, 58, 61, 81, 82, 118, 149

[19] D. R. Ghica and A. Smith. Geometry of Synthesis II: From Games to Delay-Insensitive Circuits. In *Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics (MFPS 2010)*, pages 301–324, 2010. `http://www.cs.bham.ac.uk/~drg/papers/mfps10.pdf`. vi, 28, 32, 97, 118, 138, 178, 179

[20] D. R. Ghica and A. Smith. Geometry of Synthesis III: Resource Management through Type Inference. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 345–356, New York, NY, USA, 2011. ACM. `http://www.cs.bham.ac.uk/~drg/papers/popl11.pdf`. vi, vii, 32, 59, 63, 81, 98, 103, 105, 118

[21] D. R. Ghica and A. Smith. Bounded Linear Types in a Resource Semiring. In Z. Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 331–350. Springer Berlin Heidelberg, 2014. `http://www.cs.bham.ac.uk/~drg/papers/esop14.pdf`. vii, 107, 109, 111

[22] D. R. Ghica, A. Smith, and S. Singh. Geometry of Synthesis IV: Compiling Affine Recursion into Static Hardware. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 221–233, New York, NY, USA, 2011. ACM. `http://www.cs.bham.ac.uk/~drg/papers/icfp11.pdf`. vi, 32, 153, 154

[23] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded Linear Logic. Technical Report MS-CIS-91-59, University of Pennsylvania Department of Computer and Information Sciences, 1991. `http://repository.upenn.edu/cis_reports/338/`. 6, 44, 58

[24] J. Hansen and H. Boom. The Report on the Standard Hardware Representation for ALGOL 68. *ACM SIGPLAN Notices*, 12(5):80–87, 1977. 159

[25] G. J. Holzmann. The Power of 10: Rules for Developing Safety-Critical Code. *Computer*, 39(6):95–99, June 2006. 7

[26] J. E. Hopcroft and R. M. Karp. A Linear Algorithm for Testing Equivalence of Finite Automata. Technical Report 71-114, Cornell University, 1971. `http://hdl.handle.net/1813/5958`. 58

[27] G. M. Kelly and S. MacLane. Coherence in closed categories. *Journal of Pure and Applied Algebra*, 1(1):97–140, 1971. `http://www.sciencedirect.com/science/article/pii/0022404971900132`. 49, 56, 102

[28] A. J. Kfoury. A Linearization of the Lambda-Calculus and Consequences. Technical Report BUCS-1996-021, Computer Science Department, Boston University, 1996. `http://hdl.handle.net/2144/1597`. 6, 84, 92, 93

[29] A. J. Kfoury and J. Tiuryn. Type Reconstruction in Finite Rank Fragments of the Second-Order λ-Calculus. *Information and Computation*, 98(2):228–257, 1992. 91

[30] Y. Lafont. The Finite Model Property for Various Fragments of Linear Logic. *The Journal of Symbolic Logic*, 62(4):1202–1208, 1997. 6

[31] K. Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, Department of Computer Science, New York University, 1992. `http://cs.nyu.edu/web/Research/Theses/laufer_konstantin.pdf`. 142

[32] Z. Manna and P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984. 6

[33] G. McCusker. On the Semantics of the Bad-Variable Constructor in Algol-like Languages. *Electronic Notes in Theoretical Computer Science*, 83(0):169–186, 2003. Proceedings of the 19th Conference on the Mathematical Foundations of Programming Semantics. `http://www.sciencedirect.com/science/article/pii/S1571066103500093`. 29

[34] P. W. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, July 2003. 6, 30, 134, 137, 150, 171, 173

[35] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic Control of Interference Revisited. *Electronic Notes in Theoretical Computer Science*, 1:447–486, 1995. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference. 6, 24, 139

[36] P. W. O'Hearn and D. J. Pym. The Logic of Bunched Implications. *The Bulletin of Symbolic Logic*, 5(2):215–244, 1999. 171

[37] Z. Petrić. Coherence in Substructural Categories. *Studia Logica*, 70(2):271–296, 2002. `http://arxiv.org/pdf/math.CT/0006061.pdf`. 49, 53

[38] G. Pottinger. A Type Assignment for the Strongly Normalizable λ-Terms. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry, Essays in Combinatory Logic, Lambda-Calculus and Formalism*, pages 561–577. Academic Press, 1980. 91

[39] J. C. Reynolds. Towards a Theory of Type Structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer Berlin Heidelberg, 1974. `http://repository.cmu.edu/cgi/viewcontent.cgi?article=2289&context=compsci`. 90

[40] J. C. Reynolds. Syntactic Control of Interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 39–46, 1978. `http://dl.acm.org/citation.cfm?id=512766`. 6, 30, 133, 137, 138, 150

[41] J. C. Reynolds. Syntactic control of interference, Part 2. In Ausiello, Giorgio and Dezani-Ciancaglini, Mariangiola and Della Rocca, Simonetta, editor, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer Berlin / Heidelberg, 1989. 10.1007/BFb0035793. 6, 138, 150

[42] J. C. Reynolds. The essence of ALGOL. In *ALGOL-like Languages, Volume 1*, pages 67–88. Birkhauser Boston Inc., Cambridge, MA, USA, 1997. Originally published in 1981. 6, 26

[43] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–316, 1924. `http://www.digizeitschriften.de/de/dms/img/?PPN=PPN235181684_0092&DMDID=dmdlog26`. 22

[44] A. Smith. Synthesis of imperative functions called from multiple locations. Master's thesis, University of Birmingham, 2009. 84

[45] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*. PhD thesis, Eindhoven University of Technology, 1983. `http://alexandria.tue.nl/extra1/PRF4A/8308400.pdf`. 178

[46] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. `http://dl.acm.org/citation.cfm?id=63532`. 173

[47] J. T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Eindhoven University of Technology, 1984. `http://alexandria.tue.nl/repository/books/25052.pdf`. 178

[48] A. Urquhart. Decidability and the Finite Model Property. *Journal of Philosophical Logic*, 10(3):367–370, 1981. 6

[49] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised Report on the Algorithmic Language Algol 68. *ALGOL Bulletin*, Sup 47:1–119, 1981. `http://archive.computerhistory.org/resources/text/algol/algol_bulletin/AS47/INDEX.HTM`. 25

[50] J. B. Wells. Typability and Type Checking in the Second-Order λ-calculus Are Equivalent and Undecidable. In *Proceedings of the Symposium on Logic in Computer Science*, LICS '94, pages 176–185, 1994. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.3590&rep=rep1&type=pdf`. 91