Bounded Linear Types in a Resource Semiring

Dan R. Ghica and Alex Smith

University of Birmingham, UK

Abstract. Bounded linear types have proved to be useful for automated resource analysis and control in functional programming languages. In this paper we introduce a bounded linear typing discipline on a general notion of resource which can be modeled in a semiring. For this type system we provide both a general type-inference procedure, parameterized by the decision procedure of the semiring equational theory, and a (coherent) categorical semantics. This could be a useful type-theoretic and denotational framework for resource-sensitive compilation, and it represents a generalization of several existing type systems. As a non-trivial instance, motivated by hardware compilation, we present a complex new application to calculating and controlling timing of execution in a (recursion-free) higher-order functional programming language with local store.

1 Resource-aware types and semantics

The two important things about a computer program are what it computes and what resources it needs to carry out the computation successfully. Correctness of the input-output behavior of programs has been, of course, the object of much research from various conceptual angles: logical, semantical, type-theoretical and so on. Resource analysis has been conventionally studied for algorithms, such as time and space complexity, and for programs has long been a part of research in compiler optimization.

An exciting development was the introduction of semantic [1] and especially type theoretic [14] characterizations of resource consumption in functional programming languages. Unlike algorithmic analyses, type based analysis are formal and can be statically checked for implementations of algorithms in concrete programming languages. Unlike static analysis, a typing mechanism is compositional which means that it supports, at least in principle, separate compilation and even a foreign function interface: it is an analysis based on signatures rather than implementations.

Linear logic and typing, because of the fine-grained treatment of resource-sensitive structural rules, constitute an excellent framework for resource analysis, especially in its bounded fragment [13], which can logically characterize polynomial time computation. Bounded Linear Logic (BLL) was subsequently extended to improve its flexibility while retaining poly-time [5] and further extensions to linear *dependent* typing were used to completely characterize complexity of evaluation of functional programs [4].

Such analyses use *time* as a motivating example, but can be readily adapted to other *consumable* resources such as energy or network traffic. What they have in common is a *monadic* view of resources, tracking their global usage throughout the execution of the term.

A complementary view on resource sensitivity is the *co-monadic* one, as advocated by Melliès and Tabareau [18]. The intuition is that the type system tracks how much resource a term needs in order to execute successfully. This is quite typical when controlling *reusable* resources which can be allocated and de-allocated at runtime, the typical example of which is *memory*, especially *local* (stack-allocated) memory. In fact this resource-sensitive approach is key in giving a better semantic understanding of higher-order state [17]. This view of resources is instrumental in facilitating the compilation of functional-imperative programming languages directly for resource-constrained runtimes, such as electronic circuits [8].

2 Bounded linear types over a semiring

Types are generated by the grammar $\theta ::= \sigma \mid (J{\cdot}\theta) \multimap \theta$, where σ is a fixed collection of base types and $J \in \mathcal{J}$, where $(\mathcal{J}, +, \times, \mathbf{0}, \mathbf{1})$ is a semiring. We will always take $\cdot$ to bind strongest so we will omit the brackets.

Let $\Gamma = x_1{:}J_1{\cdot}\theta_1, \ldots, x_n{:}J_n{\cdot}\theta_n$ be a list of identifiers x_i and types θ_i, annotated with semiring elements J_i. Let $fv(M)$ be the set of free variables of term M, defined in the usual way. The typing rules are:

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

$$\frac{\Gamma \vdash M : \theta}{\Gamma, x : J{\cdot}\theta' \vdash M : \theta}\ \text{Weakening}$$

$$\frac{\Gamma, x : J{\cdot}\theta \vdash M : \theta'}{\Gamma \vdash \lambda x.M : J{\cdot}\theta \multimap \theta'}\ \text{Abstraction}$$

$$\frac{\Gamma \vdash M : J{\cdot}\theta \multimap \theta' \qquad \Gamma' \vdash N : \theta}{\Gamma, J{\cdot}\Gamma' \vdash MN : \theta'}\ \text{Application}$$

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

In *Weakening* we have the side condition $x \notin fv(M)$, and in *Application* we require $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$. In the *Application* rule we use the notation

$$J{\cdot}(x_1 : K_1{\cdot}\theta_1, \ldots, x_n : K_n{\cdot}\theta_n) \triangleq x_1 : (J \times K_1){\cdot}\theta_1, \ldots, x_n : (J \times K_n){\cdot}\theta_n \qquad (1)$$

Note. For the sake of simplicity we take operations in the semiring to be resolved *syntactically* within the type system. So types such as $2{\cdot}A$ and $(1 + 1){\cdot}A$ are taken to be syntactically equal. In the context of type-checking this is reasonable because semiring actions are always constants that the type-checker can calculate with. If we were to allow resource variables, i.e. some form of resource-based polymorphism (cf. [5]) then a new structural rule would be required to handle type congruences induced by the semiring theory:

$$\frac{\Gamma, x : J\cdot\theta' \vdash M : \theta \qquad J =_{\mathcal{J}} J'}{\Gamma, x : J'\cdot\theta' \vdash M : \theta}\ \text{Semiring}$$

But in our current system this level of formalization is not worth the complication.

2.1 Examples

Bounded Linear Logic. If we take $\mathcal{J}$ to be *resource polynomials* we obtain BLL. A *monomial* is any finite product of binomial coefficients $\prod_{i=1}^{n} \binom{x_i}{n_i}$; a resource polynomial is a finite sum of monomials. They are closed under sum and product and have a semiring structure. The *Axiom* of BLL is not quite the same as ours, as we require a unit action on the type of the variable, whereas in BLL any bound can be introduced, hence the whimsical name of *Waste of Resources* for the BLL Axiom. In our system a wasteful axiom is admissible only if a resource can be decomposed as a sum involving the unit resource, by using a combination of contraction and weakening.

$$\frac{\dfrac{x : \mathbf{1}\cdot\theta \vdash x : \theta}{y : J\cdot\theta, x : \mathbf{1}\cdot\theta \vdash x : \theta}}{x : (J + \mathbf{1})\cdot\theta \vdash x : \theta}$$

The intuition of this restriction is that we need *at least* an unit of resource in order to use x.

Syntactic Control of Concurrency (SCC). It is possible to use a comonadic notion of resource to bound the number of threads used by a parallel programming language [10]. This has the advantage of identifying programs with finite-state models, with applications in automated verification [9] and in hardware synthesis [11]. If we instantiate $\mathcal{J}$ to the semiring of natural numbers we obtain SCC. However, SCC includes an additive conjunction rule to model *sequentiality*:

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

This allows to distinguish between sequential and concurrent programming language constants, e.g.: $\mathsf{seq} : !_1\cdot\mathsf{com} \times !_1\cdot\mathsf{com} \multimap \mathsf{com}$ versus $\mathsf{par} : !_1\cdot\mathsf{com} \multimap !_1\cdot\mathsf{com} \multimap \mathsf{com}$. This is an idea borrowed from Reynolds's *Syntactic Control of Interference* (SCI) [23].

This distinction between sequential and parallel composition becomes interesting when contraction is involved, e.g.

$$\lambda x.\mathsf{seq}\langle x, x\rangle : !_1\cdot\mathsf{com} \multimap \mathsf{com} \quad \text{vs.} \quad \lambda x.\mathsf{par}\, x\, x : !_2\cdot\mathsf{com} \multimap \mathsf{com}. \tag{2}$$

Note that SCC uses the notation $!_k -$ instead of $k\cdot -$ to indicate resource actions.

Tagged Control of Concurrency (TCC). SCI is akin to SCC where all bounds are set to 1. This means that in SCI the first term in Eqn. 2 can be typed, but the second cannot. Both SCI and SCC are complicated semantically by the presence of the extra additive conjunction because it lacks an adjoint exponential. The complication is also syntactic as the two composition operators have peculiarly different signatures (uncurried vs. curried).

Completing the syntactic and semantic tableau by providing both conjunctions with exponentials leads to *Bunched Typing* [21]. However, it is possible to have an SCI-like type system without using both additive and multiplicative conjunctions, but harnessing the power of an expressive enough set of resources. The elements of the semiring are a system of *tags* corresponding, intuitively, to run-time locks that need to be acquired. A notion of *safety* is introduced for tags, corresponding to the requirement that locks cannot be grabbed more than once. The restrictions on terms of an SCI-like type system can be recovered by imposing the restriction that all tags are safe. The two command compositions, sequential and parallel, have types:

$$\mathsf{seq}_{\tau_1,\tau_2} : \tau_1{\cdot}\mathsf{com} \multimap \tau_2{\cdot}\mathsf{com} \multimap \mathsf{com} \quad \text{vs.} \quad \mathsf{par}_{\tau} : \tau{\cdot}\mathsf{com} \multimap \tau{\cdot}\mathsf{com} \multimap \mathsf{com},$$

for any (safe) tags τ, τ_1, τ_2 such that $\tau_1 + \tau_2$ is also safe. Note that the two command compositions (sequential and parallel) now have the same type skeleton ($\mathsf{com} \multimap \mathsf{com} \multimap \mathsf{com}$) and no extra rules are required. The example terms in Eqn. 2 can be written in a more uniform way as:

$$\lambda x.x; x : (\tau_1 + \tau_2){\cdot}\mathsf{com} \multimap \mathsf{com} \quad \text{vs.} \quad \lambda x.x \,||\, x : (\tau + \tau){\cdot}\mathsf{com} \multimap \mathsf{com}. \tag{3}$$

As in SCI, the second one is not a valid term, as the tag $(\tau + \tau)$ cannot be safe.

The uniformity of the type skeleton is quite important for practical usage. Under the original SCI, functions that need their arguments to share information must use an uncurried signature, as opposed to functions that disallow that. A syntactic distinction that poses a sometimes difficult burden on the programmer. By contrast, in TCC the tags are inferred automatically by the compiler.

A full description of the type system, its game semantics and an application to hardware compilation is forthcoming [24].

2.2 Modularity

Given two semirings $\mathcal{J}, \mathcal{J}'$ their Cartesian product $\mathcal{J} \times \mathcal{J}'$ is also a semiring with multiplicative unit $(\mathbf{1}, \mathbf{1}')$, additive unit $(\mathbf{0}, \mathbf{0}')$ and addition and multiplication defined component-wise. Because there are many different resources one might want to track in the type system (time, space, energy, bandwidth, etc.) with significantly different properties, the fact that they can be easily combined in a modular way can be a quite appealing feature.

2.3 Type inference

We present a bound inference algorithm for the abstract system which works by creating a system of constraints to be solved, separately, by an SMT-solver that

can handle the equational theory of the resource semiring. In the type grammar, for the exponential type $J{\cdot}\theta \multimap \theta$ we allow J to stand for a concrete element of $\mathcal{J}$ or for a variable in the input program; the bound-inference algorithm will produce a set of constraints such that every model of those constraints gives rise to a typing derivation of the program without resource variables as variables are instantiated to suitable concrete values. Type judgments have form $\Gamma \vdash M : \theta \blacktriangleright \chi$, where χ is a set of equational constraints in the semiring. We also allow an arbitrary set of constants $\mathsf{k} : \theta$, which will allow the definition of concrete programming languages based on the type system. We allow each constant k to introduce arbitrary resource constraints χ_{k}

$$\frac{}{x : \mathbf{1}{\cdot}\theta \vdash x : \theta \blacktriangleright \mathit{true}} \qquad \frac{}{\emptyset \vdash \mathsf{k} : \theta \blacktriangleright \chi_{\mathsf{k}}}$$

$$\frac{\Gamma \vdash M : \theta \blacktriangleright \chi}{\Gamma, x : J{\cdot}\theta' \vdash M : \theta \blacktriangleright \chi} \qquad \frac{\Gamma, x : J{\cdot}\theta \vdash M : \theta' \blacktriangleright \chi}{\Gamma \vdash \lambda x : \theta.M : J{\cdot}\theta \multimap \theta' \blacktriangleright \chi}$$

$$\frac{\Gamma, x : J_1{\cdot}\theta', y : J_2{\cdot}\theta'' \vdash M : \theta \blacktriangleright \chi}{\Gamma, x : J{\cdot}\theta' \vdash M[x/y] : \theta \blacktriangleright \chi \cup \{J = J_1 + J_2\} \cup \overline{\theta' = \theta''}}$$

$$\frac{\Gamma \vdash M{:}J{\cdot}\theta \multimap \theta' \blacktriangleright \chi \qquad x_1{:}J_1{\cdot}\theta_1, \ldots, x_n{:}J_n{\cdot}\theta_n \vdash N{:}\theta'' \blacktriangleright \chi'}{\Gamma, x_1{:}J_1'{\cdot}\theta_1, \ldots, x_n{:}J_n'{\cdot}\theta_n \vdash MN{:}\theta' \blacktriangleright \chi \cup \chi' \cup \{J_k' = J{\cdot}J_k \mid k = 1, n\} \cup \overline{\theta = \theta''}}$$

The constraints of shape $\overline{\theta_1 = \theta_2}$ are to be interpreted in the obvious way, as the set of pairwise equalities between resource bounds used in the same position in the two types:

$$\overline{\sigma = \sigma} \stackrel{def}{=} \emptyset$$

$$\overline{J_1{\cdot}\theta_1 \multimap \theta_1' = J_2{\cdot}\theta_2 \multimap \theta_2'} \stackrel{def}{=} \{J_1 = J_2\} \cup \overline{\theta_1 = \theta_2} \cup \overline{\theta_1' = \theta_2'}.$$

If $\mathcal{M}$ is a model, i.e. a function mapping variables to concrete values, by $\Gamma[\mathcal{M}]$ we write the textual substitution of each variable by its concrete value in a sequent. The following is then true by construction:

Theorem 1. *If $\Gamma \vdash M : \theta \blacktriangleright \chi$ and $\mathcal{M}$ is a model of the system of constraints χ in the semiring $\mathcal{J}$ then $(\Gamma \vdash M : \theta)[\mathcal{M}]$ is derivable.*

2.4 Categorical semantics

We give an abstract framework suitable for interpreting the abstract type system of Sec. 2. Up to this point the calling discipline of the type system was not relevant, as there are no side-effects, but for giving an interpretation we need to make this choice. In order to remain relevant to our motivating application, hardware compilation, we shall choose the *call-by-name* mechanism, which is used by the *Geometry of Synthesis* compiler.

We require two categories. We interpret *computations* in a symmetric monoidal closed category $(\mathcal{G}, \otimes, I)$ in which the tensor unit I is a terminal object. Let α be the *associator* and λ, ρ be the right and left *unitors*. We write the unique morphism into the terminal object as $!_A : A \to I$. Currying is the isomorphism

$$\Lambda_{A,B,C} : A \otimes B \to C \simeq A \to B \multimap C,$$

and the evaluation morphism is $eval_{A,B} : A \otimes (A \multimap B) \to B$.

We interpret *resources* in a category $\mathcal{R}$ with two monoidal tensors $(\oplus, 0)$ and $(\odot, 1)$ such that:

$$\begin{aligned} J \odot (K \oplus L) &\simeq J \odot K \oplus J \odot L && \text{(r-distributivity)} \\ (J \oplus K) \odot L &\simeq J \odot L \oplus K \odot L && \text{(l-distributivity)} \\ J \odot 0 &\simeq 0 \odot J \simeq 0 && \text{(zero).} \end{aligned}$$

The action of resources on computations is modeled by a functor $\cdot : \mathcal{R} \times \mathcal{G} \to \mathcal{G}$ such that the following natural isomorphisms must exist:

$$\delta_{J,K,A} : J{\cdot}A \otimes K{\cdot}A \simeq (J \oplus K){\cdot}A \tag{4}$$

$$\pi_{R,R',A} : R{\cdot}(R'{\cdot}A) \simeq (R \odot R'){\cdot}A \tag{5}$$

$$\zeta_A : 0{\cdot}A \simeq I \tag{6}$$

$$\iota_A : \mathbf{1}{\cdot}A \simeq A \tag{7}$$

and the following diagrams commute:

$$\begin{array}{ccc} J{\cdot}A \otimes K{\cdot}A \otimes L{\cdot}A & \xrightarrow{\delta_{J,K,A} \otimes 1_{L\cdot A}} & (J \oplus K){\cdot}A \otimes L{\cdot}A \\ \downarrow{\scriptstyle 1_{J\cdot A} \otimes \delta_{K,L,A}} & & \downarrow{\scriptstyle \delta_{J\oplus K,L,A}} \\ J{\cdot}A \otimes (K \oplus L){\cdot}A & \xrightarrow{\delta_{J,K\oplus L,A}} & (J \oplus K \oplus L){\cdot}A \end{array} \tag{8}$$

$$\begin{array}{ccc} J{\cdot}A \otimes K{\cdot}A & \xrightarrow{\delta_{J,K,A}} & (J \oplus K){\cdot}A \\ \downarrow{\scriptstyle J\cdot f \otimes K\cdot f} & & \downarrow{\scriptstyle (J\oplus K)\cdot f} \\ J{\cdot}B \otimes K{\cdot}B & \xrightarrow{\delta_{J,K,B}} & (J \oplus K){\cdot}B \end{array} \tag{9}$$

Natural isomorphism π (Eqn. 5) reduces successive resource actions on computations to a composite resource action, corresponding to the product of the semiring. Natural isomorphism $\delta_{J,K,A}$ in Eqn. 4 is a "quantitative" version of the diagonal morphism in a Cartesian category, which collects the resources of the contracted objects. The commuting diagram in Eqn. 8 stipulates that the order in which we use the "quantitative" diagonal order to contract several objects is irrelevant, and the commuting diagram in Eqn. 9 gives a "quantitative" counterpart for the naturality of the diagonal morphism. Finally, Eqns. 6 and 7 show the connection between the units of the tensors involved.

A direct consequence of the naturality of ρ and I being terminal, useful for proving coherence, is:

Proposition 1. *The following diagram commutes in the category $\mathcal{G}$ for any $f : B \to C$:*

$$\begin{array}{ccccc} B \otimes A & \xrightarrow{1_B \otimes !_A} & B \otimes I & \xrightarrow{\rho_B} & B \\ \downarrow{\scriptstyle f \otimes 1_A} & & & & \downarrow{\scriptstyle f} \\ C \otimes A & \xrightarrow{1_C \otimes !_A} & C \otimes I & \xrightarrow{\rho_C} & C. \end{array}$$

Computations are interpreted in a canonical way in the category $\mathcal{G}$. Types are interpreted as objects and terms as morphisms, with

$$[\![J{\cdot}\theta \multimap \theta']\!]_{\mathcal{G}} = ([\![J]\!]_{\mathcal{R}}{\cdot}[\![\theta]\!]_{\mathcal{G}}) \multimap [\![\theta']\!]_{\mathcal{G}}.$$

From now on, the interpretation of the resource action is written as J instead of $[\![J]\!]_{\mathcal{R}}$ when there is no ambiguity and the subscript of $[\![-]\!]_{\mathcal{G}}$ is left implicit.

Environments are interpreted as

$$[\![\Gamma]\!] = [\![x_1 : J_1{\cdot}\theta_1, \ldots x_n : J_n{\cdot}\theta_n]\!] = J_1{\cdot}[\![\theta_1]\!] \otimes \cdots \otimes J_n{\cdot}[\![\theta_n]\!].$$

Terms are morphisms in $\mathcal{G}$, $[\![\Gamma \vdash M : \theta]\!]$ defined as follows:

$$[\![x : \mathbf{1}{\cdot}\theta \vdash x : \theta]\!] = \iota_{[\![\theta]\!]}$$
$$[\![\Gamma, x : J{\cdot}\theta \vdash M : \theta']\!] = 1_{[\![\Gamma]\!]} \otimes !_{J\cdot[\![\theta]\!]}; \rho_{[\![\Gamma]\!]}; [\![\Gamma \vdash M : \theta]\!]$$
$$[\![\Gamma \vdash \lambda x.M : J{\cdot}\theta \multimap \theta']\!] = \Lambda_{J\cdot[\![\theta]\!]}\big([\![\Gamma, x : J{\cdot}\theta \vdash M : \theta']\!]\big)$$
$$[\![\Gamma, J{\cdot}\Gamma' \vdash FM : \theta']\!] = ([\![\Gamma \vdash F : J{\cdot}\theta \multimap \theta']\!] \otimes J{\cdot}[\![\Gamma' \vdash M : \theta]\!]); \mathit{eval}_{J\cdot[\![\theta]\!],[\![\theta']\!]}$$
$$[\![\Gamma, x : (J + K){\cdot}\theta \vdash M[x/y] : \theta']\!] = 1_{[\![\Gamma]\!]} \otimes \delta_{J,K,\theta}; [\![\Gamma, x : J{\cdot}\theta, y : K{\cdot}\theta \vdash M : \theta]\!].$$

2.5 Coherence

The main result of this section is the coherence of typing. The derivation trees are not unique because there is choice in the use of the weakening and contraction rules. Since meaning is calculated on a particular derivation tree we need to show that it is independent of it. The coherence conditions for the monoidal category are standard [15], but what is interesting and new is that resource manipulation does not break coherence. The key role is played by the isomorphism δ which is the resource-sensitive version of contraction, which can combine or de-compose resources without loss of information.

The key idea of the proof is that we can bring any derivation tree to a standard form (which we call *stratified*), with weakening and contraction performed as late as possible. A combination of weakenings and contractions can bring a term to linear form, which has a uniquely determined derivation tree. The key result is Lem. 3 which stipulates that the order in which contractions and weakenings are performed is irrelevant.

The following derivation rules is admissible because it is a chain of contractions and weakenings, followed by an abstraction:

$$\frac{x_1 : J_1{\cdot}\theta, \ldots, x_n : J_n{\cdot}\theta, \Gamma \vdash M : \theta'}{\Gamma \vdash \lambda x.M[x/x_i] : (J_1 + \cdots + J_m){\cdot}\theta \multimap \theta'}\ \mathrm{ACW}$$

where $x, x_j \notin fv(M)$, for some $1 \leq m \leq n$, and all $1 \leq i \leq m$, $m \leq j \leq n$. Variables $x_1, \ldots, x_m$ are contracted into a fresh variable x and dummy variables $x_{m+1}, \ldots, x_n$ can be added.

We denote sequents $\Gamma \vdash M : \theta$ by Σ and derivation trees by ∇. Let $\Lambda(\Sigma) \in \{id, wk, ab, ap, co, acw\}$ be a label on the sequents, indicating whether a sequent

is derived using the rule for identity, weakening, etc. If a sequent $\Sigma = \Gamma \vdash M : \theta$ is the root of a derivation tree ∇ we write it Σ^{∇} or $\Gamma \vdash^{\nabla} M : \theta$.

We say that a sequent is *linear* if each variable in the environment Γ occurs freely in the term M exactly once.

Definition 1. *For a linear sequent, we call a* stratified *derivation tree the unique derivation tree produced by the following deterministic algorithm.*

- MN*: The only possible rule is* Application *and, since the judgement* $\Gamma, J \cdot \Delta \vdash MN : \theta$ *is about a linear term, both* $\Gamma \vdash M : J \cdot \theta' \multimap \theta$ *and* $\Delta \vdash N : \theta'$ *are linear and there is only one way* Γ *can be split, unless* $J = 0$*. In this case any resource actions in* Δ *can be chosen, since they will be zeroed by the action of* J*. To keep the algorithm deterministic we choose zeroes. This ensures that every resource action in the derivation of* N *is also 0.*
- $\lambda x.M$*: We use* AWC *to give each occurrence of* $x : J \cdot A$ *in* M *a new (fresh) name* $x_i : J_i : A$*. Each* J_i *is uniquely determined by the context in which* x_i *occurs. Note that it is necessary that* $\sum J_i \leq J$*, otherwise the term cannot be typed.*
- x*: The only possible rule is* Weakening*.*

Lemma 1. *If a linear sequent has a derivation tree then it has a (unique) stratified derivation tree. Moreover, all the sequents occurring in the tree are linear.*

Proof. The proof is almost immediate (by contradiction). Linear derivations cannot use weakening or contractions except where they can be replaced by AWC, so to construct a stratified tree, we just ned to normalise uses of 0.

We now show that any derivation can be reduced to a stratified derivation through applying a series of meaning-preserving tree transformations, which we call *stratifying rules*.

The Weakening rule commutes trivially with all other rules except Identity, Abstraction and Contraction, if they act on the weakened variable. In thes cases we replace the sequence of Weakening followed by Abstraction and/or Contraction with the combined *AWC* rule. The more interesting tree transformation rules are for Contraction.

Contraction commutes with Application. There are two pairs of such rules, one for pushing down contraction in the function and one for pushing down contraction in the argument:

$$\dfrac{\dfrac{\Gamma, x : J\cdot\theta, y : J'\cdot\theta \vdash F : J_1\cdot\theta_1 \multimap \theta_2}{\Gamma, x : (J + J')\cdot\theta \vdash F[x/y] : J_1\cdot\theta_1 \multimap \theta_2} \qquad \Gamma' \vdash M : \theta_1}{\Gamma, x : (J + J')\cdot\theta, J_1\cdot\Gamma' \vdash F[x/y]M : \theta_2}$$

$$\Longleftrightarrow^{AL}$$

$$\dfrac{\dfrac{\Gamma, x : J\cdot\theta, y : J'\cdot\theta \vdash F : J_1\cdot\theta_1 \multimap \theta_2 \qquad \Gamma' \vdash M : \theta_1}{\Gamma, x : J\cdot\theta, y : J'\cdot\theta, J_1\cdot\Gamma' \vdash FM : \theta_2}}{\Gamma, x : (J + J')\cdot\theta, J_1\cdot\Gamma' \vdash (FM)[x/y] : \theta_2}$$

Similarly for pushing down contraction from the argument side and similarly for rules involving weakening:

$$\frac{\Gamma \vdash F : J_1 \cdot \theta_1 \multimap \theta_2 \qquad \dfrac{\Gamma, x : J \cdot \theta, y : J' \cdot \theta \vdash M : \theta_1}{\Gamma, x : (J + J') \cdot \theta \vdash M[x/y] : \theta_1}}{\Gamma, x : (J_1 \times (J + J')) \cdot \theta, \Gamma' \vdash F(M[x/y]) : \theta_2}$$

$$\overset{AR}{\Longleftrightarrow}$$

$$\frac{\dfrac{\Gamma \vdash F : J_1 \cdot \theta_1 \multimap \theta_2 \qquad \Gamma', x : J \cdot \theta, y : J' \cdot \theta \vdash M : \theta_1}{\Gamma, J_1 \cdot \Gamma', x : (J_1 \times J) \cdot \theta, y : (J_1 \times J') \cdot \theta \vdash FM : \theta_2}}{\Gamma, x : (J_1 \times J + J_1 \times J') \cdot \theta, \Gamma' \vdash (FM)[x/y] : \theta_2}$$

Contraction also commutes with Abstraction, if the contracted and abstracted variables are distinct, $x \neq y$:

$$\frac{\dfrac{\dfrac{\Gamma, x : J \cdot \theta, x' : J' \cdot \theta, y : K \cdot \theta' \vdash M : \theta''}{\Gamma, x : (J + J') \cdot \theta, y : K \cdot \theta' \vdash M[x/x'] : \theta''}}{\Gamma, x : (J + J') \cdot \theta \vdash \lambda y.M[x/x'] : K \cdot \theta' \multimap \theta''}}{}$$

$$\overset{CA}{\Longleftrightarrow}$$

$$\frac{\dfrac{\dfrac{\Gamma, x : J \cdot \theta, x' : J' \cdot \theta, y : K \cdot \theta' \vdash M : \theta''}{\Gamma, x : J, x' : J' \cdot \theta \vdash \lambda y.M : K \cdot \theta' \multimap \theta''}}{\Gamma, x : (J + J') \cdot \theta \vdash (\lambda y.M)[x/x'] : K \cdot \theta' \multimap \theta''}}{}$$

The rule for swapping contraction and weakening is (types are obvious and we elide them for concision):

$$\frac{\dfrac{\Gamma, y, z \vdash M}{\Gamma, y \vdash M[y/z]}}{\Gamma, y, x \vdash M[y/z]} \overset{WC}{\Longleftrightarrow} \frac{\dfrac{\Gamma, y, z \vdash M}{\Gamma, y, z, x \vdash M}}{\Gamma, y, x \vdash M[y/z]}$$

The final rule is to zero-out the resource actions of free identifiers used in derivations of functions with zero-types.

$$\frac{\Gamma \vdash M : 0 \cdot \theta \multimap \theta' \qquad \Gamma' \vdash N : \theta}{\Gamma, 0 \cdot \Gamma' \vdash MN : \theta'} \overset{ZO}{\Longleftrightarrow} \frac{\Gamma \vdash M : 0 \cdot \theta \multimap \theta' \qquad 0 \cdot \Gamma' \vdash N : \theta}{\Gamma, 0 \cdot \Gamma' \vdash MN : \theta'}$$

Proposition 2. *The following judgments are syntactically equal*

$$\begin{aligned}
&\Gamma, x : \theta, \Gamma' \vdash F[x/y]M : \theta' = \Gamma, x : \theta, \Gamma' \vdash (FM)[x/y] : \theta', \\
&\Gamma, x : (J_1 \times (J + J')) \cdot \theta, \Gamma' \vdash F(M[x/y]) : \theta_2 \\
&\qquad = \Gamma, x : (J_1 \times J + J_1 \times J') \cdot \theta, \Gamma' \vdash (FM)[x/y] : \theta_2, \\
&\Gamma, x : (J + J') \cdot \theta \vdash \lambda y.M[x/x'] : K \cdot \theta' \multimap \theta' \\
&\qquad = \Gamma, x : (J + J') \cdot \theta \vdash (\lambda y.M)[x/x'] : K \cdot \theta' \multimap \theta''.
\end{aligned}$$

Proof. The proof of the first two statements is similar. Because Application is linear it means that an identifier y occurs either in F or in M, but not in both.

Therefore $(FM)[x/y]$ is either $F(M[x/y])$ or $(F[x/y])M$. This makes the terms syntactically equal. In any semiring, $J_1 \times (J + J') = J_1 \times J + J_1 \times J'$, which makes the environments equal. Note that semiring equations are resolved syntactically in the type system, as pointed out at the beginning of this section. For the third statement we know that $x \neq y$.

Proposition 3. *If ∇ is a derivation and ∇' is a tree obtained by applying a stratifying rule then ∇' is a valid derivation with the same root $\Sigma^{\nabla} = \Sigma^{\nabla'}$ and the same leaves.*

Proof. By inspecting the rules and using Prop. 2.

Most importantly, stratifying transformation preserve the meaning of the sequent.

Lemma 2. *If $\nabla \Rightarrow \nabla'$ is a stratifying rule then $[\![\Sigma^{\nabla}]\!] = [\![\Sigma^{\nabla'}]\!]$.*

Proof. By inspecting the rules. Prop. 3 states that the root sequents are equal and the trees are well-formed. For WC (and the other rules involving the stratification of Weakening) this is an immediate consequence of Prop. 1. For AL and AR the equality of the two sides is an immediate consequence of symmetry in $\mathcal{G}$ and the functoriality of the tensor $\otimes$. For CA the equality of the two sides is an instance of the general property in a symmetric monoidal closed category that $f; \Lambda(g) = \Lambda((f \otimes 1_{B'}); g)$ for any $A \xrightarrow{f} B$, $B \otimes B' \xrightarrow{g} C$. For ZO the equality is given by the (zero) isomorphism in the resource category and the ζ isomorphism (Eqn. 6).

Lemma 3. *If ∇, ∇' are derivation trees consisting only of* Contraction *and* Weakening *with a common root Σ then $[\![\Sigma^{\nabla}]\!] = [\![\Sigma^{\nabla'}]\!]$.*

Proof. Weakening commutes with any other rule (Prop. 1). Changing the order of multiple contraction of the same variable uses the associativity coherence property in Eqn. 8. Changing the order in which different variables are contracted uses the naturality coherence property in Eqn. 9.

The lemma above ensures that the AWC rule is itself semantically coherent.

Lemma 4. *If ∇ is a derivation there exists a stratified derivation tree ∇' which can be obtained from ∇ by applying a (finite) sequence of stratifying tree transformations. Moreover, $[\![\Sigma^{\nabla}]\!] = [\![\Sigma^{\nabla'}]\!]$.*

Proof. The stratifying transformations push contraction and weakening through any other rules and the derivation trees have finite height. If a contraction or weakening cannot be pushed through a rule it means that the rule is an abstraction on the variable being contracted or weakened, and we replace the rules with AWC. For the weakening and contractions pushed to the bottom of the tree the order is irrelevant, according to Lem. 3 The result is a stratified tree. Next we apply induction on the chain of stratifying rules using Lem. 2 for every rule application and Lem. 3 for the final chain of weakening and contractions.

Theorem 2 (Coherence). *For any derivation trees ∇_1, ∇_2 with common root Σ, $[\![\Sigma^{\nabla_1}]\!] = [\![\Sigma^{\nabla_2}]\!]$.*

Proof. Using Lem. 4, ∇_1, ∇_2 must be effectively stratifiable into trees ∇'_1, ∇'_2 with the same root and $[\![\Sigma^{\nabla_i}]\!] = [\![\Sigma^{\nabla'_i}]\!]$ for $i = 1, 2$. We first reduce Σ^{∇_i} to a linear form (using contractions and weakenings) then use Lem. 1. The only difference between ∇'_1, ∇'_2 are the order of the abstractions and permutations at the bottom of the tree, and the choice of names of variables, both of which are semantically irrelevant (Lem. 3).

3 Case study: timing analysis

In the sequel we will present a more complex resource semiring which we shall use in giving a precise type-level analysis of timing. The interpretation of the type $J{\cdot}\theta \multimap \theta'$ is that the function needs a *schedule of execution* J for the argument in order to execute. Again, note the comonadic interpretation of resources. This type system is interesting in its own right, as a way of capturing timing at the level of the type system. A full blown analysis for timing bounds, as part of a more general approach to certifying resource bounds, has been given before using dependent types [3]. However, this approach only automates the *certification* of the bounds whereas we fully automate the process, at the expense of less precision.

A *schedule* $J = [x_1, x_2, \ldots, x_n]$ is a multiset of *stages* x_i, which are one-dimensional contractive affine transformations over $\mathbb{R}$. This means that our reading of time is a *relative* one. A *contractive* affine transformation is represented as $x_{s,p} = \begin{pmatrix} s & p \\ 0 & 1 \end{pmatrix}$, where $0 \leq s \leq 1$ and $0 \leq s + p \leq 1$.. The value s is a *scaling factor* relative to the unit interval, and p is a *phase change*, i.e. a delay from the time origin. For example, $x_{.25,.5} = \begin{pmatrix} .25 & .5 \\ 0 & 1 \end{pmatrix}$ represents a stage that starts when $\frac{1}{2}$ of the duration has elapsed and lasts for $\frac{1}{4}$ the duration relative to which we are measuring. Some extreme values are $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ which overlaps perfectly to the reference interval or $\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$ which starts at the end of the reference interval and has zero duration (is instantaneous).

For an example of how schedules are interpreted as type annotations, the type $[x_{.5,0}, x_{.5,.5}]{\cdot}\mathsf{com} \multimap \mathsf{com}$ is of a function that executes its argument twice. First argument starts instantly and the second starts half-way through its execution; both take $\frac{1}{2}$ of the execution.

In mathematical terms, schedules are the *semigroup semiring of one-dimensional contractive affine transformations*, usually written as $\mathcal{J} = \mathbb{N}[\mathrm{Aff}_1^c]$. This is a canonical construction which has the mathematical properties we desire.

Contractive affine transformations enable composition of timed functions in a natural way, because such transformations compose, by matrix product. Com-

posing time represented as absolute intervals is perhaps possible, but it complicates the rules of the type system significantly. By using relative timing the rules of the system are clean, at the expense of having a rather complicated final step of elaborating relative into absolute timings for a closed term (i.e. a program), as it will be seen in Sec. 3.3.

When we refer to the timing of a computation, and it is unambiguous from context, we will sometimes use just x to refer to its action on the unit interval $u = [0, 1]$. For example, if we write $x \subseteq x'$ we mean $x{\cdot}u \subseteq x'{\cdot}u$, i.e. $[p, s + p] \subseteq [p', s' + p']$, i.e. $p \geq p'$ and $s + p \leq s' + p'$. If we write $x \leq x'$ we mean the Egli-Milner order on the two intervals, $x{\cdot}u \leq x'{\cdot}u$, i.e. $p \leq p'$ and $s + p \leq s' + p'$. If we write $x \cap x' = \emptyset$ we mean the two intervals are disjoint, $x{\cdot}u \cap x'{\cdot}u = \emptyset$, etc.

Contractive affine transformations form a semigroup with matrix product as multiplication and unit element $I \triangleq \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. The semiring of a semigroup $(\mathcal{G}, \times, I)$ is a natural construction from any semiring and any semigroup. In our case the semiring is natural numbers ($\mathbb{N}$), so the semigroup semiring is the set of finitely supported functions $J : \mathrm{Aff}_1^c \to \mathbb{N}$ with

$$\mathbf{0}(x) = 0 \qquad \mathbf{1}(x) = \begin{cases} 1 & \text{if } x = I \\ 0 & \text{otherwise} \end{cases}$$

$$(J + K)(x) = J(x) + K(x) \qquad (J \times K)(x) = \sum_{\substack{y,z \in \mathrm{Aff}_1^c \\ y \times z = x}} J(y) \times K(z).$$

This is isomorphic to finite multisets over Aff_1^c. We use interchangeably whichever representation is more convenient.

3.1 A concrete programming language

A concrete programming language is obtained by adding a family of functional constants in the style of Idealized Algol [22]. We take commands and integer expressions as the base types, $\sigma ::= \mathsf{com} \mid \mathsf{exp}$.

Ground-type constants are just $n : \mathsf{exp}$ and $\mathsf{skip} : \mathsf{com}$. Ground-type operators are provided with explicit timing information. For example, for commands we have a family of timed composition operators (i.e. schedulers):

$$\mathsf{comp}_{x,y} : [x]{\cdot}\mathsf{com} \multimap [y]{\cdot}\mathsf{com} \multimap \mathsf{com}.$$

Both sequential and parallel composition are subsumed by the timed scheduler. Sequential composition is a scheduler in which the arguments are non-overlapping, with the first argument completing before the second argument starts: $\mathsf{seq}_{x,y} = \mathsf{comp}_{x,y}$ where $x \leq y$ and $x \cap y = \emptyset$ (which we write $x < y$). Parallel composition is simply $\mathsf{par}_x = \mathsf{comp}_{x,x}$, with both arguments initiating and completing execution at the same time. Schedulers that are neither purely sequential nor parallel, but a combination thereof, are also possible.

Arithmetic operators and branching (if) are also given explicit timings.

$$\mathsf{op}_{x,y} : [x]{\cdot}\mathsf{exp} \multimap [y]{\cdot}\mathsf{exp} \multimap \mathsf{exp},$$
$$\mathsf{if}_{x,y} : [x]{\cdot}\mathsf{exp} \multimap [y]{\cdot}\sigma \multimap [y]{\cdot}\sigma \multimap \sigma, \quad x < y.$$

Note that branching has an additional sequentiality constraint which stipulates that the guard must execute before the branches are allowed to start executing. This is not a type-related constraint, but a language-level constraint.

Assignable variables are handled by separating read and write access, as is common in Idealized Algol (IA). Let the type of *acceptors* be defined (syntactically) as the familly $\mathsf{acc}_w \triangleq [w]{\cdot}\mathsf{exp} \multimap \mathsf{com}$. There is no stand-alone var type, instead the reader and writers to a variable are bound to the same memory location by a block variable constructor with signature:

$$\mathsf{new}_{\sigma,J,w_1,\ldots,w_n} : (J{\cdot}\mathsf{exp} \multimap \mathsf{acc}_{w_1} \multimap \cdots \multimap \mathsf{acc}_{w_n} \multimap \sigma) \multimap \sigma, \quad \sigma \in \{\mathsf{exp}, \mathsf{com}\}.$$

The asymmetric treatment of readers and acceptors is a consequence of using call-by-name: the read operation is an expression thunk with no arguments, but the acceptor needs to evaluate its argument which can take an arbitrary amount of time. For programmer convenience var-typed identifiers can be sugared into the language but, because the read and write schedules of access need to be maintained separately, the contraction rules become complicated (yet routine) so we omit them here.

Example 1. The timings of the IA program $\mathsf{new}\, v.\, v := !v + 1$ can be captured by this typing system. First let us write it in a functional-style syntax where the occurrences of v are linearized: $\mathsf{new}(\lambda v_1 \lambda v_2 . v_2(\mathsf{add}\, v_1\, 1))$. The type of this linearized local-variable binder is $\mathsf{new} : (\mathsf{exp} \multimap \mathsf{acc} \multimap \mathsf{com}) \multimap \mathsf{com}$.

The next step is to determine schedules of execution for the constants. The typing derivation is

$$\dfrac{\dfrac{\dfrac{v_2{:}[w]{\cdot}\mathsf{exp}\multimap\mathsf{com} \vdash v_2{:}\mathsf{acc}_w \qquad \dfrac{\dfrac{v_1{:}\mathsf{exp} \vdash v_1{:}\mathsf{exp} \qquad \vdash \mathsf{add}_{x,y}{:}[x]{\cdot}\mathsf{exp}\multimap[y]{\cdot}\mathsf{exp}\multimap\mathsf{exp}}{v_1{:}[x]{\cdot}\mathsf{exp} \vdash \mathsf{add}_{x,y}\, v_1{:}[y]{\cdot}\mathsf{exp}\multimap\mathsf{exp}} \qquad \vdash 1{:}\mathsf{exp}}{v_1{:}[x]{\cdot}\mathsf{exp} \vdash \mathsf{add}_{x,y}\, v_1\, 1{:}\mathsf{exp}}}{v_2{:}\mathsf{acc}_w, v_1{:}[w\times x]{\cdot}\mathsf{exp} \vdash v_2(\mathsf{add}_{x,y}\, v_1\, 1){:}\mathsf{com}}}{\vdash \lambda v_1 \lambda v_2 . v_2(\mathsf{add}_{x,y}\, v_1\, 1){:}[w\times x]{\cdot}\mathsf{exp}\multimap\mathsf{acc}_w\multimap\mathsf{com}}$$

for any stages x, y, w. To complete the term we need to apply the binder $\mathsf{new}_{\mathsf{com},[w\times x],w}$. Written in a fully sugared notation, this term would be: $\mathsf{new}_{\mathsf{com},[w\times x],w}\, x := !x +_{x,y} 1$. We will see later how to choose sensible concrete values for the stages.

3.2 Type inference for pipelining

Computing such detailed timings can perhaps be useful when doing real-time computation using programs with higher-order functions without recursion, as this language is expressive enough for implementing, for example, certain digital

signal processing algorithms. However, we will look at a different application motivated by hardware compilation: imposing a pipelining discipline via the type system. Pipelining is important because it allows the concurrent use of a hardware component and thus reduces the overall footprint of a program compiled in hardware. Without it, any concurrently used component is systematically replicated, a process called *serialization* [11].

The constraints imposed by the typing system, as seen in Example 1 can be quite loose, and there can be broad choice in selecting concrete values for the stages. In some sense this is a bug, because there can be no principal type, but we will turn it into a handy feature by introducing extra constraints motivated by the *platform* to which we are compiling the program, in this case one relying on pipelining. Thus the overall system of constraints will contain *type*, *language* and *platform* constraints independently of each other, a pleasant degree of modularity. The rest of the section describes the type inference algorithm.

First an observation: the general recipe from Sec. 2.3 cannot be immediately applied because there is no (off-the-shelf) SMT solver for $\mathbb{N}[\mathrm{Aff}_1^c]$. We need to run the SMT in two stages: first we calculate the sizes of the multiset (as in SCC inference), which allows us to reduce constraints in $\mathbb{N}[\mathrm{Aff}_1^c]$ to constraints in Aff_1^c. Then we map equations over Aff_1^c into real-number equations, which can be handled by the SMT solver. There is a final, bureaucratic, step of reconstructing the multi-sets from the real-number values. To fully automate the process we start with the Hindley-Milner type inference to determine the underlying simple-type structure [19].

Multiset size (SCC) type inference is presented in detail elsewhere [11], but we will quickly review it here. We first interpret schedules as natural numbers, representing their number of stages $J \in \mathbb{N}$. Unknown schedules are variables, schedules with unknown stages but fixed size (such as those for operators) are constants. A type derivation results in a constraint system over $\mathbb{N}$ which can be solved by an SMT tool such as Z3 [20]. More precisely, Z3 can attempt to solve the system, but it can be either unsatisfiable in some cases or unsolvable as nonlinear systems of constraints over $\mathbb{N}$ are generally undecidable.

As a practical observation, solving this constraint using general-purpose tools will give an arbitrary solution, if it exists, whereas a "small" solution is preferable. [11] gives a special-purpose algorithm guaranteed to produce solutions that are in a certain sense minimal. To achieve a small solution when using Z3 we set a global maximum bound which we increment on iterated calls to Z3 until the system is satisfied.

Next we instantiate schedules to their known sizes, and to re-run the inference algorithm, this time in order to compute the stages. This proceeds according to the general type-inference recipe, resulting in a system of constraints over the $\mathbb{N}[\mathrm{Aff}_1^c]$ semiring, with the particular feature that all the sizes of all the multisets is known. We only need to specify the schedules for the constants:

$$\frac{}{\vdash 1 : \mathsf{exp} \blacktriangleright \mathit{true}} \qquad \frac{}{\vdash \mathsf{skip} : \mathsf{com} \blacktriangleright \mathit{true}}$$

$$\frac{}{\vdash \mathsf{op}_{x,y} : [x]\cdot\sigma \multimap [y]\cdot\sigma \multimap \sigma \blacktriangleright \{x \neq I, y \neq I\}}$$

$$\vdash \mathsf{if}_{x,y} : [x]\cdot\mathsf{exp} \multimap [y]\cdot\sigma \multimap [y]\cdot\sigma \multimap \sigma \blacktriangleright \{x < y\}$$

$$\vdash \mathsf{new}_{\sigma,J,w_1,\ldots,w_n} : (J\cdot\mathsf{exp} \multimap \mathsf{acc}_{w_1} \multimap \cdots \multimap \mathsf{acc}_{w_n} \multimap \sigma) \multimap \sigma \blacktriangleright \bigwedge_{i=1,n}\{0 \neq w_i\}$$

In the typing for op we disallow an instant response and in the typing for new we disallow instantanewous write operations.

As mentioned, in the concrete system it is useful to characterize the resource usage of families of constants also by using constraints, which can be combined with the other constraints (of the type system, etc.). The language of constraints itself can be extended arbitrarily, provided that eventually we can represent it into the language of our external SMT solver, Z3. The constraints introduced by the language constants are motivated as follows:

op: We prevent the execution of any of the two arguments to take the full interval, because an arithmetic operation cannot be computed instantaneously.
if: The execution of the guard must precede that of the branches.
new: The write-actions cannot be instantaneous.

This allows us to translate the constraints from the semiring theory into real-number constraints. Solving the system (using Z3) gives precise timing bounds for all types. However, this does not guarantee the fact that computations can be pipelined, it just establishes timings. In order to force a pipeline-compatible timing discipline we need to add extra constraints guaranteeing the fact that each timing annotation J is in fact a proper pipeline.

Two stages $x_1, x_2 \in \mathrm{Aff}_1^c$ are *FIFO* if they are Egli-Milner-ordered, $x_1 \leq x_2$. They are *strictly FIFO*, written $x_1 \lhd x_2$ if they are FIFO and they do not start or end at the same time, i.e. if $x_i\cdot[0,1] = [t_i, t_i']$ then $t_0 \neq t_0'$ and $t_1 \neq t_1'$.

Definition 2. *We say that a schedule $J \in \mathbb{N}[\mathrm{Aff}_1^c]$ is a* pipeline*, written* $\mathsf{Pipe}(J)$*, if and only if $\forall x \in \mathrm{Aff}_1^c, J(x) \leq 1$ (i.e. J is a proper set) and for all $x, x' \in J$, either $x \lhd x'$ or $x' \lhd x$ or $x = x'$.*

Given a system of constraints χ over $\mathbb{N}[\mathrm{Aff}_1^c]$, before solving it we augment it with the condition that every schedule is a proper pipeline: for any J used in χ, $\mathsf{Pipe}(J)$. Using the conventional representation (scaling and phase), the usual matrix operations and the pipelining definitions above we can represent χ as a system of constraints over $\mathbb{R}$, and solve it using Z3.

Implementation note. For the implementation, we enforce arbitrary orders on the stages of the pipeline and, if that particular order is not satisfiable then a different (arbitrary) order is chosen and the process is repeated. However, spelling out the constraint for the existence of a pipelining order $\lhd$ for any schedule J would entail a disjunction over all possible such orders, which is $\mathcal{O}(n!)$ in the size of the schedule, for each schedule, therefore not realistic. However, if the systems of constraints have few constants and mostly unknowns, i.e. we are trying to find a schedule rather than accommodate complex known schedules, our experience shows that this pragmatic approach is reasonable.

Example 2. Let us first consider the simple problem of using three parallel adders to compute the sum $fx + fx + fx + fx$ when we know the timings of f. Suppose $f : ([(0.5, 0.1); (0.5, 0.2)]\cdot\mathsf{exp} \multimap \mathsf{exp}$, i.e. it is a two-stage pipeline where the execution of the argument takes half the time of the overall execution and have relative delays of 0.1 and 0.2 respectively. We have the choice of using three adders with distinct schedules $+_i : [x_i]\cdot\mathsf{exp} \multimap [y_i]\cdot\mathsf{exp} \multimap \mathsf{exp}$ $(i \in \{1, 2, 3\})$ so that the expression respects the pipelined schedule of execution of f. The way the operators are associated is relevant: $(fx +_2 fx) +_1 (fx +_3 fx)$. Also note that part of the specification of the problem entails that the adders are trivial (single-stage) pipelines. Following the algorithm above, the typing constraints are resolved to the following:

$$
\begin{aligned}
+_1 &: [(0.5, 0.265625)]\cdot\mathsf{exp} \multimap [(0.5, 0.25)]\cdot\mathsf{exp} \multimap \mathsf{exp} \\
+_2 &: [(0.5, 0.21875)]\cdot\mathsf{exp} \multimap [(0.5, 0.25)]\cdot\mathsf{exp} \multimap \mathsf{exp} \\
+_3 &: [(0.5, 0.375)]\cdot\mathsf{exp} \multimap [(0.5, 0.25)]\cdot\mathsf{exp} \multimap \mathsf{exp}
\end{aligned}
$$

In the implementation, the system of constraints has 142 variables and 357 assertions, and is solved by Z3 in circa 0.1 seconds on a high-end desktop machine.

Example 3. Let us now consider a more complex, higher-order example. Suppose we want to calculate the convolution $(*)$ of a pipelined function $(f : [(0.5, 0.1); (0.5, 0.2)]\cdot\mathsf{exp} \multimap \mathsf{exp})$ with itself four times. And also suppose that we want to use just two instances of the convolution operator $*_1, *_2$, so we need to perform contraction on it as well. The type skeleton of the convolution operator is $(*) : (\mathsf{exp} \to \mathsf{exp}) \to (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp} \to \mathsf{exp}$.

The implementation of f and $*$ are unknown, so we want to compute the timings for the term

$$
\begin{aligned}
(*_1) &: J_1^{vi}\cdot(J_1^{i}\cdot(J_1^{ii}\cdot\mathsf{exp}\multimap\mathsf{exp}) \to J_1^{iv}\cdot(J_1^{iii}\cdot\mathsf{exp}\multimap\mathsf{exp})\multimap J_1^{v}\cdot\mathsf{exp}\multimap\mathsf{exp}), \\
(*_2) &: J_2^{vi}\cdot(J_2^{i}\cdot(J_2^{ii}\cdot\mathsf{exp}\multimap\mathsf{exp}) \to J_2^{iv}\cdot(J_2^{iii}\cdot\mathsf{exp}\multimap\mathsf{exp})\multimap J_2^{v}\cdot\mathsf{exp}\multimap\mathsf{exp}), \\
f &: J_3\cdot([(0.5, 0.1); (0.5, 0.2)]\cdot\mathsf{exp}\multimap\mathsf{exp}) \vdash (f *_1 f) *_2 (f *_1 f) : \theta.
\end{aligned}
$$

The constraint system has 114 variables and 548 assertions and is solved by Z3 in 0.2 seconds on a high-end desktop machine. The results are:

$$
\begin{aligned}
J_1^i = J_1^{iv} = J_2^i = J_2^{iv} &= [(1.0, 0.0)] \\
J_1^{ii} = J_1^{iii} = J_1^v = J_2^{ii} = J_2^{iii} = J_2^v &= [(0.5, 0.1); (0.5, 0.2)] \\
J_1^{vi} = J_3 &= [(0.5, 0.125); (0.5, 0.25); (0.5, 0.375); (0.5, 0.4375)] \\
J_2^{vi} &= [(0.25, 0.25); (0.25, 0.5); (0.25, 0.625)]
\end{aligned}
$$

3.3 Absolute timing

This section is a variation of the type system in order to deal with absolute rather than relative timing. The presentation is more informal, but the formalism of the previous sections can be applied here if desired.

In our main intended application, hardware compilation, relative timing rather than absolute timing is relevant. However, for other applications such as real-time computing absolute timing might be required. We can recover absolute timings for a program (closed term) in two steps. What is interesting here is the introduction of yet another level of constraints, this times imposed by the physical characteristics of the computational platform we use. They come in addition to the structural, language and architectural constraints seen so far.

In the first step we propagate the timing annotations all the way down to the constants. The constants of the language are families indexed by schedules, and this propagation will generate the set of all concrete constants used by a program, with timings given relative to the overall execution of the program. The function $\ulcorner - \urcorner$ takes as arguments a schedule and a term and produces as set of language constants. It is defined inductively on the type derivation as follows:

$$\begin{aligned}
\ulcorner x : \mathbf{1}\cdot\theta \vdash x : \theta\urcorner(J) &= \emptyset \\
\ulcorner \Gamma, x : K\cdot\theta \vdash M : \theta'\urcorner(J) &= \ulcorner \Gamma \vdash M : \theta'\urcorner(J), \quad x \notin fv(M) \\
\ulcorner \Gamma \vdash \lambda x.M : K\cdot\theta \multimap \theta'\urcorner(J) &= \ulcorner \Gamma, x : K\cdot\theta \vdash M : \theta'\urcorner(J) \\
\ulcorner \Gamma, K\cdot\Gamma' \vdash FM : \theta'\urcorner(J) &= \ulcorner \Gamma \vdash F : K\cdot\theta \multimap \theta'\urcorner(J) \cup \ulcorner \Delta \vdash M : \theta\urcorner(J \times K) \\
\ulcorner \Gamma, x : (K + L)\cdot\theta \vdash M[x/y] : \theta'\urcorner(J) &= \ulcorner \Gamma, x : K\cdot\theta, y : L\cdot\theta \vdash M : \theta'\urcorner(J) \\
\ulcorner \mathsf{k} : \theta\urcorner(J) &= \{\mathsf{k} : \ulcorner\theta\urcorner([x]) \mid x \in J\} \\
\ulcorner K \cdot \theta \multimap \theta'\urcorner(J) &= K \cdot \ulcorner\theta\urcorner(J) \multimap \ulcorner\theta'\urcorner(J) \\
\ulcorner \sigma\urcorner(J) &= J \cdot \sigma.
\end{aligned}$$

What is the most interesting is the translation of the constants. In the case of our concrete programming language we have, for example:

$$\ulcorner \mathsf{op} : [x]\cdot\mathsf{exp} \multimap [y]\cdot\mathsf{exp} \multimap \mathsf{exp}\urcorner[u] = \mathsf{op} : [u \times x]\cdot\mathsf{exp} \multimap [u \times y]\cdot\mathsf{exp} \multimap [u]\cdot\mathsf{exp}$$

and so on. This is a constant which executes in interval u, and its arguments in $u \times x$ and $u \times y$, which now represent absolute timings. The reasons that we collect these constants is because depending on the concrete target platform some of them may be impossible to implement from a timing point of view. For the operation above (op), if we work out the numbers we get $t_1 = u_1x_1 + u_1x_2 + u_2$ and $t_2 = u_1y_1 + u_1y_2 + u_2$ as the respective times when the two arguments terminate, which means that the duration in which op must compute the result is before its own termination at $t = u_1 + u_2$, i.e. $\delta t = u_1 - \max(u_1x_1 + u_1x_2, u_1y_1 + u_1y_2)$. This δt must be greater than a system-defined constant such as the duration of one clock cycle (e.g. 1 ns).

For any program $\vdash M : \mathsf{com}$, its set of constants is $\ulcorner \vdash M : \sigma\urcorner([d])$ where d is an affine (not necessarily contractive) transform defining its total duration. The value of d is not known and must be chosen large enough so that all constants in $\ulcorner M\urcorner[d]$ are implementable.

Example 4. Consider the term $\vdash 1+_{x,y}(2+_{u,v}3) : \mathsf{exp}$. It is quite easy to calculate that

$$\begin{aligned}\ulcorner\vdash 1 +_{x,y} (2 +_{u,v} 3) : \mathsf{exp}\urcorner([d]) = \{&+ : [d \times x] \cdot \mathsf{exp} \multimap [d \times y] \cdot \mathsf{exp} \multimap [d] \cdot \mathsf{exp},\\ &+ : [d \times x \times u] \cdot \mathsf{exp} \multimap [d \times x \times v] \cdot \mathsf{exp} \multimap [d \times x] \cdot \mathsf{exp},\\ &1 : [d \times x] \cdot \mathsf{exp}, 1 : [d \times x \times u] \cdot \mathsf{exp}, 1 : [d \times x \times v] \cdot \mathsf{exp}\}\end{aligned}$$

Suppose that all the additions can performed in 1 ns and the constants can be computed instantaneously. These timing constraints are satisfied by $d = \begin{pmatrix}2 & 0\\ 0 & 1\end{pmatrix}$, $y = \begin{pmatrix}0.5 & 0\\ 0 & 1\end{pmatrix}$, and $x, u, v = \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}$.

4 Related work

The BLL type system has been already generalized by Dal Lago and collaborators to Linear Dependent Types (LDT) [4, 6]. This greatly increases the expressiveness of the type system but at the expense of losing decidability. We also generalize BLL but in a different way, by using an abstract notion of resource. It is natural to think of resources as having a monoidal structure, as resources can be *aggregated*. However, we show that the additional structure of a semiring can be employed in a useful way to *scale* resources. Our generalization consists of replacing the family of modalities $!_x A$ of BLL, which are interpreted as A *may be reused less than* x *times* with a general resource action $R{\cdot}A$, which is interpreted as A *may use at most* R *resources*. This is a generalization because R can be simply instantiated to x, giving back BLL. For this abstract type system we show how the problem of type inference can be naturally reduced to a system of constraints parametrized by the equational theory of the resource semiring. Provided this theory is decidable, a type inference algorithm automatically follows.

We also provide a categorical framework, for which we prove the key result of *coherence*. This is the main technical contribution of the paper. Coherence is an essential technical property because denotational interpretations are given inductively on type derivations, which are generally not unique. This means that in the absence of coherence a denotational interpretation cannot make sense. Coherence for a categorical semantics is also the generalization of the subject reduction property used by operational semantics, as substitution is usually interpreted by composition in the category. Resource-awareness has been usually modeled operationally, but game-semantic [7] and, more recently, relational models [16] have been proposed to model resources denotationally.

The same typing framework presented here was developed independently in [2] (published in this volume), but includes resource actions in covariant positions so it could be used to model call-by-value languages. For this larger type system the soundness of the system is proved relative to an operational semantics with so-called *coeffect actions*.

The second part of the paper presents a non-trivial motivating application to timing analysis and automated pipelining of computations in a recursion-free functional programming language with local store, and is meant to illustrate several points. The first one is showing a complex notion of resource in action. The second one is presenting a non-trivial multi-stage type inference algorithm for this resource. The third one is to show a specialization of the type inference algorithm in the case of a concrete programming language when language constants and arbitrary system-level resources can come into play.

5 Conclusion

We have presented an abstract framework for BLL using a more general notion of resource which can be modeled in a semiring, gave a categorical model and a proof of coherence. We gave several instances of this general typing framework, depending on several notions of resource, one of which is a fairly elaborate method for tracking execution time in a higher-order setting. We have not given concrete semantics here, but denotational (game) models of various programming languages that fit this framework have been developed elsewhere [10, 12, 24].

One methodological feature which seems quite unique for this typing framework, and is amply illustrated in the previous section, is its degree of flexibility and modularity. In addition to the structural constraints imposed by the type system we can freely add language-level constraints (e.g. "the if statement is sequential"), architectural constraints (e.g. "schedules must be pipelines") and physical constraints (e.g. "addition can be performed no faster than 1 ns"). Various passes of the type-inference algorithm collect constraints which, ultimately, are about what language constants are implementable or not within certain resource constraints on a particular physical platform. The modularity of the system is expressed in a different dimension as well. Since the Cartesian product of semirings is a semiring we can easily combine unrelated notions of constraints, which is essential in managing the trade-offs that need to be made in a realistic system.

Acknowledgment. Sec. 2.4 benefited significantly from discussions with Steve Vickers. Olle Fredriksson and Fredrik Nordvall Forsberg provided useful comments. The authors express gratitude for their contribution.

References

1. Gérard Boudol. The lambda-calculus with multiplicities (abstract). In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 1993.
2. Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In *ESOP*, 2014.
3. Karl Crary and Stephanie Weirich. Resource bound certification. In *POPL 2000*, pages 184–198, New York, NY, USA, 2000. ACM.

4. Ugo Dal Lago and Marco Gaboardi. Linear Dependent Types and Relative Completeness. *Logical Methods in Computer Science*, 8(4), 2011.
5. Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. In Pierre-Louis Curien, editor, *TLCA*, volume 5608 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.
6. Ugo Dal Lago and Barbara Petit. The geometry of types. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 167–178. ACM, 2013.
7. Dan R. Ghica. Slot games: a quantitative model of computation. In *POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 85–97. ACM, 2005.
8. Dan R. Ghica. Geometry of Synthesis: a structured approach to VLSI design. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 363–375. ACM, 2007.
9. Dan R. Ghica and Andrzej S. Murawski. Compositional model extraction for higher-order concurrent programs. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2006.
10. Dan R. Ghica, Andrzej S. Murawski, and C.-H. Luke Ong. Syntactic control of concurrency. *Theor. Comput. Sci.*, 350(2-3):234–251, 2006.
11. Dan R. Ghica and Alex Smith. Geometry of synthesis iii: resource management through type inference. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 345–356. ACM, 2011.
12. Dan R. Ghica and Alex Smith. From bounded affine types to automatic timing analysis. *CoRR*, abs/1307.2473, 2013.
13. J.Y. Girard, A. Scedrov, and P.J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical computer science*, 97(1):1–66, 1992.
14. Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *LICS*, pages 464–473. IEEE Computer Society, 1999.
15. G.M Kelly. On MacLane's conditions for coherence of natural associativities, commutativities, etc. *Journal of Algebra*, 1(4):397 – 402, 1964.
16. Jim Laird, Giulio Manzonetto, Guy McCusker, and Michele Pagani. Weighted relational models of typed lambda-calculi. In *LICS*, pages 301–310. IEEE Computer Society, 2013.
17. Paul-André Melliès and Nicolas Tabareau. An algebraic account of references in game semantics. *Electr. Notes Theor. Comput. Sci.*, 249:377–405, 2009.
18. Paul-André Melliès and Nicolas Tabareau. Resource modalities in tensor logic. *Ann. Pure Appl. Logic*, 161(5):632–653, 2010.
19. R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
20. Leonardo Moura and Nikolaj Bjrner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.
21. Peter W. O'Hearn. On bunched typing. *J. Funct. Program.*, 13(4):747–796, 2003.
22. J.C. Reynolds. The essence of ALGOL. In *ALGOL-like Languages, Volume 1*, pages 67–88. Birkhauser Boston Inc., 1997.
23. John C. Reynolds. Syntactic control of interference. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *POPL*, pages 39–46. ACM Press, 1978.
24. Alexander Smith. *Type-directed hardware synthesis*. PhD thesis, University of Birmingham, forthcoming.