Efficient Parallelism Through Ergometric, Multilinear Futures

Aditi Gupta

Advisor: Prof. Frank Pfenning, Computer Science Department

Motivation

Goal: allow programmers to write **efficient parallel code** in a functional language without significant overheads.

Prior work has focused primarily on fork-join programs, in which computations split into branches and later synchronize. Here, we instead consider a form of parallelism known as **futures**, which are more general but more difficult to implement efficiently. Futures involve spawning multiple threads of computation that execute simultaneously.

However, the efficiency of parallel programs is limited by a few key factors:

1. **Granularity**: the cost of starting and scheduling parallel computations can outweigh the benefits, especially on inexpensive computations. A programmer must decide when it is worthwhile to compute something in parallel.
2. **Garbage Collection**: prior work suggests that one of the bottlenecks to efficient parallelism is in the garbage collector – specifically, how memory gets "cleaned up" once it is done being used.

We address these challenges by considering a linear, resource-aware system inspired by session types, and then generalize to programs without linearity restrictions. In our language, linear, multilinear, and nonlinear types coexist to reap the benefits of linearity.

This language serves as **a basis for an efficient, general, and expressive implementation of functional futures**.

Background Information

Parallelism. When multiple computations do not rely on one another, we can run them at the same time (instead of waiting for one to finish before starting the other). Many algorithms have substantially lower cost (i.e., they run faster) when run in parallel.

Futures. A future begins a computation and immediately moves on to the next steps of a program. This allows a program to simultaneously compute both the expression in the future and the next instructions without waiting for the earlier expression to finish.

Ergometric Types. We can augment our type system with ergometric types to track (statically) the cost that will be incurred at a given step; in executing that step, we use up some amount of "potential." Without sufficient potential, a program will fail to typecheck.

Linearity. Linearity enforces a restriction that each piece of data must be used exactly once. This severely restricts the programs we can write, but offers efficiency benefits for futures.

Outline of Research

1. Experiment with futures in SML and in Rast [4].
2. Generalize to mixed linear/nonlinear framework.
3. Add reference counting to allow most programs to be written multilinearly, including potential.

Linear Futures

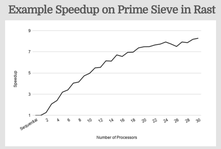
Linearity can improve efficiency for parallel programs with futures. We experimented with both SML and Rast (a linear, resource-aware, session-typed language).

Asymptotic Benefits:
- Linear futures can improve the theoretical asymptotic efficiency of pipelined programs [3].

Garbage Collection:
- Linear data can be easily garbage collected, since memory can be deallocated immediately after use [8].

Granularity Control:
- With ergometric types, we can calculate work at the type level and automate granularity control decisions.
- Based on preliminary experimentation in Rast, we can aggregate the work of multiple sequentially-executing processes to identify when to execute in parallel, and thus achieve speedups on pipelined programs

Example Speedup on Prime Sieve in Rast



Summary of Results (Max Speedup vs. Sequential)

	Program	Speedup
SML	Summing 50,000 consecutive numbers	x2.20
	Prime sieve up to 10,000	x2.51
	Merging two depth-20 binary trees	x7.93
	Quicksort on length 100 list	x1.76
Rast	Summing 8,192 consecutive numbers	x4.18
	Prime sieve up to 2,048	x8.29
	Inserting 70 elements into a trie	x2.45

Conclusion: We don't want to enforce a linearity restriction, because it's infeasible to write purely linear programs, but we still want linearity benefits.

SML	Rast	New Language
Nonlinear: practical for a wide variety of programs	Linearity restriction	Adjoint – linear and shared modes, with reference counting
Manual granularity control for each computation	Ergometric types: automated granularity control	Potential annotations for multilinear programs
Challenging garbage collection	Easy garbage collection	Easy garbage collection for multilinear programs
Futures implemented on top of original language	Natural concurrency, built into the language	Natural concurrency, built into the language

A Linear/Nonlinear Type System

- 2 modes: *l*, linear, and *u*, unrestricted [2]. Unrestricted data can't rely on linear data or have potential.

$$
\begin{array}{rcll}
A_m, B_m & ::= & \mathbf{1}_m & \text{true} & \text{unit} \\
& | & \oplus_m\{l : A^l_m\}_{l\in L} & \text{disjunction} & \text{variant records} \\
& | & \&_m\{l : A^l_m\}_{l\in L} & \text{additive conjunction} & \text{lazy records} \\
& | & A_m \otimes_m B_m & \text{multiplicative conj.} & \text{eager pairs} \\
& | & A_m \multimap_m B_m & \text{implication} & \text{functions} \\
& | & \uparrow A_{\mathsf{L}}\ (m = \mathsf{U}) & \text{shift L to U} & \text{coerce to nonlinear} \\
& | & \downarrow A_{\mathsf{U}}\ (m = \mathsf{L}) & \text{shift U to L} & \text{coerce to linear} \\
& | & \triangleright^r A_{\mathsf{L}}\ (m = \mathsf{L}) & \text{pay potential} & \text{store potential in type} \\
& | & \triangleleft^r A_{\mathsf{L}}\ (m = \mathsf{L}) & \text{get potential} & \text{harvest potential in type} \\
& | & t & & \text{defined (recursive) type}
\end{array}
$$

- Processes read from and write to typed addresses.
- The state of a system is represented by a configuration, containing cells (holding values or continuations) and threads to execute.
- We annotate typing judgments and dynamics steps with potential required.

Multilinearity

Limitation of Adjoint System: for any data that is unrestricted, we enjoy no linearity benefits => treat as much of the language as possible as linear, only using the unrestricted mode when strictly necessary

- We can inductively copy or drop any purely positive, linear cell or a shifted shared cell: *multilinear types*
- Instead of using shared cells, in most cases, we can just use a linear cell and copy/drop it as necessary
- Because inductive copying/dropping is expensive and tedious, introduce **reference counting**: addresses can have multiple clients, which share potential
  - Many variables can refer to the same address; substitutions are stored in closing environments
- Define splitting and dropping definitions for multilinear types
- Still allows easy garbage collection, precise potential annotations, and algorithmic advantages => benefits of linearity without restrictions

$$
\begin{aligned}
&\mathsf{cell}(c, V, n), \mathsf{thread}(d, q, [\eta, c/x], \mathsf{alias}\ x : \tau_1\ \mathsf{as}\ y : \tau_2, z : \tau_3\ ;\ P) \\
&\qquad \longmapsto\ \mathsf{cell}(c, V, n+1), \mathsf{thread}(d, q, [\eta, c/y, c/z], P) \\
&\mathsf{cell}(c, V, n), \mathsf{thread}(d, q, [\eta, c/x], \mathsf{drop}\ x : \tau\ ;\ P) \\
&\qquad \longmapsto\ \mathsf{cell}(c, V, n-1), \mathsf{thread}(d, q, [\eta], P)
\end{aligned}
$$

Dynamics for aliasing and dropping: modify reference count and update substitution closure η

Invariants
- Reference count = number of clients
- Total potential in cell = sum of potential seen by clients

Safety

Theorem 1 (Progress). *If $\cdot \Vdash^q \mathcal{C} :: \Psi$, then either $\mathcal{C}$ final or $\mathcal{C} \longmapsto^{q'} \mathcal{C}'$ for some configuration $\mathcal{C}'$ and $q' \leq q$.*

A valid configuration is either complete, with no more threads, or takes a step to a new configuration (and has sufficient potential to do so).

Theorem 2 (Preservation). *If $\Psi \Vdash^{q+w} \mathcal{C} :: \Psi'$, and $\mathcal{C} \longmapsto^{w} \mathcal{C}'$ for some configuration $\mathcal{C}'$, then $\Psi \Vdash^{q} \mathcal{C}' :: \Psi'$.*

When a configuration takes a step (using up some potential), the result remains well-typed, providing the same addresses as the original.

Conclusion

We have presented a core language in which linear, multilinear, and nonlinear types coexist to achieve the benefits of linearity without its restrictions.

We explored the advantages of linear futures through experimentation in SML and Rast, including algorithmic speedups with pipelining, garbage collection, and granularity control. Then, we developed a type system that begins to reconcile practicality with efficient parallelism.

This language retains the linear facets of Rast, including ergometric types, but also allows for non-linear programs. We added reference counting by distinguishing between addresses and variables and introducing a way to split potential. This allows us to write most programs purely within the multilinear setting and enjoy the benefits of linearity.

We developed the statics and dynamics of such a language and proved its safety (progress and preservation).

Practical Use and Future Directions

There are many avenues of further research that we hope to explore, largely centered around the usability of the language described.

- **Implementation:** Implementing this language would allow us to demonstrate its use practically; we have not yet considered the details of granularity in an adjoint setting and would need to experiment with running actual programs.
- **Surface Syntax:** This is intended as an intermediate language: we anticipate that we will be able to compile a functional language down to this language (with most operations remaining sequential, but some occurring in parallel), and then compile this language to machine code. We leave the details of the surface syntax to future work.
  - Cost annotations could be generated automatically from a source language in the style of RAML [9] or partially reconstructed from a source language in the style of Rast.
- **Automation:** Ideally, we will eventually be able to automatically infer when copy/drop needs to be called so as to prevent users from having to identify these locations manually.
  - We hope to re-introduce arithmetic refinements, which exist in Rast but which we omitted here for simplicity. This would allow us to track additional information about data structures that provides a more precise understanding of potential.
  - We might wish to explore ways of further automating granularity control; in our current work, users still have to experiment with grain values manually. For instance, we can experiment with machine learning [6] or oracle-guided [1] techniques.

Selected References

[1] Umut Acar et al. "Provably and practically efficient granularity control". PPoPP 2019.

[2] P. N. Benton. "A Mixed Linear and Non-Linear Logic". CSL 1994.

[3] Guy Blelloch and Margaret Reid-Miller. "Pipelining with Futures". TCS 1999.

[4] Ankush Das and Frank Pfenning. "Rast: Resource-Aware Session Types with Arithmetic Refinements". FSCD 2020.

[5] Ankush Das, Jan Hoffmann, and Frank Pfenning. "Work Analysis with Resource-Aware Session Types". LICS 2018.

[6] Ankush Das and Jan Hoffmann. "ML for ML: learning cost semantics by experiment". TACAS 2017.

[7] Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. "Semi-Axiomatic Sequent Calculus". FCSD 2020.

[8] Jean-Yves Girard and Yves Lafont. "Linear Logic and Lazy Computation". TAPSOFT 1987.

[9] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Resource Aware ML". CAV 2012.

[10] Klaas Pruiksma and Frank Pfenning. "Back to Futures". JFP 2022.