# Extended Abstract: Affine killing

## Semantics for stopping the ParT

Kiko Fernandez-Reyes
Uppsala University
kiko.fernandez@it.uu.se

Dave Clarke
Uppsala University
dave.clarke@it.uu.se

## Abstract

Speculative, parallel abstractions allow that, once a result is computed, the remaining (unnecessary) speculative computations can be safely stopped. However, it is difficult to know when it is safe to stop an ongoing computation. This paper presents a refinement of the parallel speculative ParT abstraction [3] with an affine type system that allows *in-place updates*, and *killing* speculative computations using *thread-local reasoning*. There is ongoing work to prove the soundness of the calculus and implement it in the Encore language [1].

***CCS Concepts*** • **Theory of computation → Functional constructs**; **Type structures**; *Parallel computing models*; *Type theory*;

***Keywords*** type systems, concurrency, tasks, parallelism, speculative parallelism, concurrency

**ACM Reference format:**
Kiko Fernandez-Reyes and Dave Clarke. 2017. Extended Abstract: Affine killing. In *Proceedings of , Oxford, United Kingdom, September 2017 (TyDe'17),* 3 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Parallel languages, such as Encore [1], can spawn potentially millions of parallel computations; each spawned computation returns a future, a placeholder for the result when the spawned computation finishes. Without high-level abstractions, the creation of complex coordination workflows using futures becomes a difficult task, e.g. the creation and coordination of pipeline parallelism handling thousand of tasks and killing speculative computations in such setting is not trivial. ParT [3] is a speculative, parallel abstraction that simplifies the process of spawning parallel tasks and killing unnecessary computations. Values and ongoing parallel computations are lifted to the ParT abstraction; the programmer controls this abstraction via combinators (explained later). Consider the following example, in Encore, which uses combinators from the ParT parallel abstraction:

```
1   class Facebook
2     def findInfoFb(user: User): Info
3       ...
4     end
5   end
6   fun findFriend(user: User, t: Twitter,
7                                fb: Facebook): Par[Info]
8     val twInfo = {t} >> (fun x => x.findInfoTw(user))
9     val fbInfo = {fb} >> (fun x => x.findInfoFb(user))
10    twInfo >> updateDB
11    getPhoneNumber << (twInfo || fbInfo)
12  end
```

This code performs an asynchronous parallel search of a person's name on two different social networks. The example starts by lifting values to the ParT abstraction (lines 8 and 9, `{t}` and `{fb}`). The anonymous functions use the asynchronous and parallel map combinator ($\gg :: Par[t] \to (t \to t') \to Par[t']$), which asynchronously applies the function given as second argument to the first argument, returning immediately a new ParT on which more operations can be done, such as saving the result into a database as soon as the information is available (line 10). Next, the composition combinator ($\| :: Par[t] \to Par[t] \to Par[t]$) produces a new ParT that groups the ParTs `twInfo` and `fbInfo` (line 11). Afterwards, the prune combinator ($\ll :: (Fut[Maybe[t]] \to Par[t']) \to Par[t] \to Par[t']$) takes two arguments, a function and a ParT with ongoing computations and returns a new ParT abstraction. The function starts immediately and its first argument represents the value of the first computation that finishes from the ParT (given as second argument to the prune combinator), if any. In this case, prune selects the first result returned by the computations from lines 8 and 9, and apply the function `getPhoneNumber` to the result, killing the remaining computations as they are no longer needed. However, the variable `twInfo` has two aliases (lines 10 and 11) and, if the `fbInfo` computation finishes before the one from Twitter, the ParT abstraction should not merrily kill the ongoing Twitter computation – other computations depend on `twInfo`.

Our work leverages static information from an affine type system to safely kill computations using *thread-local reasoning* and *optimise* the ParT abstraction – currently, we do not handle side-effects in speculative computations.

## 2 Affine type systems

An affine type system allows values to be used once or in an unrestricted manner [4]. The example given above could potentially be encoded as:

```
class Facebook
  def findInfoFb(user: read User): lin Info
    ...
  end
end

fun findFriend(user: read User, t: read Twitter,
                fb: read Facebook): lin Par[lin Info]
  ...
```

Partly funded by the EU project FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations.

*TyDe'17, Oxford, United Kingdom*
© 2017 ACM. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

```
10  end
```

This refined example adds affine annotations `lin` and `read` (lines 2, 7 and 8) to indicate whether the variables can be aliased, where `lin` does not allow aliasing but `read` allows unrestricted aliasing.

With these annotations in place, an affine type system gives *static aliasing guarantees* that can be *exploited by parallel speculative abstractions.*

## 3 Core idea

Speculative, parallel abstractions can leverage the static information of affine type systems and use *thread-local reasoning* to safely kill ongoing speculative computations.

We define an affine type system that uses type-directed elaboration rules to add affine annotations to parallel combinators (shown in Section 1); the specialised affine combinators are implemented to exploit linearity (if present). We also make the composition ( || ) combinator polymorphic and define a subtype affine relation, `lin` <: `read` , so that a linear ParT can contain linear and unrestricted references but not the other way around – as that would be unsound. Thus, the type-directed elaboration rule for the composition combinator (in line 11) proceeds as:

$$\frac{\Delta;\Gamma_1 \vdash \texttt{twInfo} \hookrightarrow \texttt{twInfo}' : \kappa_1\ Par[T] \qquad \Delta;\Gamma_2 \vdash \texttt{fbInfo} \hookrightarrow \texttt{fbInfo}' : \kappa_2\ Par[T]}{\Delta;\Gamma_1\ \Gamma_2 \vdash \texttt{twInfo} \,||\, \texttt{fbInfo} \hookrightarrow \texttt{twInfo}' \,||_{\texttt{lin}}\ \texttt{fbInfo}' : \texttt{lin}\ Par[T]}$$

where $\Delta$ represents the unrestricted environment, $\Gamma_1\ \Gamma_2$ the linear one (with $\Gamma_1 \cap \Gamma_2 = \varnothing$), $\kappa_1 = \texttt{read}$ (since `twInfo` is aliased, lines 8 and 10) and $\kappa_2 = \texttt{lin}$ (line 2 from the affine example).

The example above, after the type-directed elaboration rules, ends up with the following runtime annotations:

$$\Delta;\Gamma_1\ \Gamma_2 \vdash \{\texttt{twInfo}\}_{\texttt{read}}\ ||_{\texttt{lin}}\ \{\texttt{fbInfo}\}_{\texttt{lin}}\ : \texttt{lin}\ Par[T]$$

These annotations enable a kind of *dynamic dispatch* based on the structure of the ParT. All combinators benefit from it, performing (at least) in-place updates [5] whenever deemed safe. For example, the map combinator ($\gg$) applied to the current example produces new values, re-using the memory allocated for the linear singleton structure in $\{\texttt{fbInfo}\}_{\texttt{lin}}$ and allocating new memory for the new ParT resulting from the other one (as $\{\texttt{twInfo}\}_{\texttt{read}}$ can be aliased).

The prune combinator further exploits the static information and dynamic dispatch to *decide* which computations can be *safely killed.* Figure 1 is a pictographical representation of the example so far; the `twInfo` ParT has multiple dependencies (namely the update of the database and the possibility to fetch a phone number). These dependencies – statically catched by the affine mode – prevent the runtime from killing its underlaying ongoing computations as that would be unsound. The case for the `fbInfo` ParT is different: the type system ensures that this computation does not have *dependencies* and it is aliased-free – its result can only be used in a single place – making it a safe candidate to kill. In this example, if the Twitter computation finishes before the one from Facebook, it is easy to see how killing the ongoing Facebook computation cannot have any effect on other computations – we can apply *thread-local reasoning* to killing the Facebook computation. Below are the runtime rules, for this scenario, for pruning and killing computations

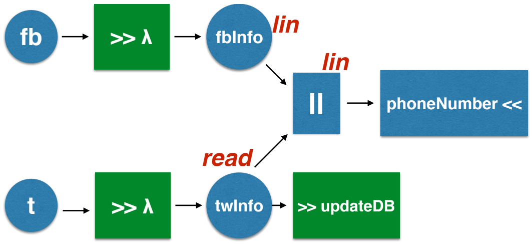

**Figure 1.** Pictographical representation of example in Introduction

(we have abbreviated `twInfo` to `t` and `fbInfo` to `b`):

[PRUNE]

$$(\texttt{task}_g\ E[v \ll_{\texttt{lin}}\ (\{\texttt{t}\}_{\texttt{read}}\ ||_{\texttt{lin}}\ \{\texttt{b}\}_{\texttt{lin}}\ )]) \rightarrow$$
$$(\texttt{task}_f\ (\texttt{peek}_{\texttt{lin}}\ \{\texttt{t}\}_{\texttt{read}}\ ||_{\texttt{lin}}\ \{\texttt{b}\}_{\texttt{lin}}\ ))\ (\texttt{fut}_f)\ (\texttt{task}_g\ E[v\ f])$$

[LINEAR PEEKING]

$$(\texttt{task}_g\ (\texttt{peek}_{\texttt{lin}}\ (\{t\}_{\texttt{read}}\ ||_{\texttt{lin}}\ \{b\}_{\texttt{lin}}\ ))) \rightarrow$$
$$(\texttt{task}_g\ (\texttt{Just t}))\ (\texttt{kill}\ g) \bigcup_{h\,\in\,\texttt{linDeps}(b)} (\texttt{kill}\ h)$$

[KILL]

$$(\texttt{task}_g\ b)\ (\texttt{kill}\ g) \rightarrow (\texttt{kill}\ g) \bigcup_{h\,\in\,\texttt{linDeps}(b)} (\texttt{kill}\ h)$$

The prune combinator relies on the hidden runtime function `peek` [3] that kills the speculative computations (rule LINEAR PEEKING) applying thread-local reasoning: safely traversing asynchronous computations spawned by the `fbInfo` (`linDeps(b)`) and killing them (rule KILL). Therefore, our approach to killing computations *respects* unrestricted computations and kills the linear ones.

## 4 Related work

One approach [6, 7] to safely killed speculative computations relies on adding well-defined safe-points, defined by the programmer, where it is safe to stop a speculative computation. This approach puts more responsibility on the programmer, who has to specify the number and position of these checkpoints. Other approach [8] performs a privatisation of their address space to allow safe-independent mutation. This is not necessary in our setting since we are dealing with a functional language and the affine type system takes care of the aliasing problem. Our previous approach [3] was formalised such as it dynamically tracks the dependencies among parallel computations in the ParT, relying on a global view of the system that determines when a computation does not have any more dependencies. In terms of implementation, the initial design creates a runtime representation of connections between ParTs, represented as a directed acyclic graph (*DAG*). Each node in the graph represents a singleton value and, upon executing the prune combinator, the runtime traverses the *DAG* checking which computations have more than one forward connection, i.e., a ParT used more than once by different computations. This approach was never implemented due to the high implementation complexity of the runtime and garbage collection protocol [2].

## References

[1] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse

at the Parallel Language Encore. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures (Lecture Notes in Computer Science)*, Marco Bernardo and Einar Broch Johnsen (Eds.), Vol. 9104. Springer, 1–56. DOI: https://doi.org/10.1007/978-3-319-18941-3_1

[2] Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully concurrent garbage collection of actors on many-core machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 553–570. DOI: https://doi.org/10.1145/2509136.2509557

[3] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. 2016. ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations. In *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings (Lecture Notes in Computer Science)*, Alberto Lluch-Lafuente and José Proença (Eds.), Vol. 9686. Springer, 101–120. DOI: https://doi.org/10.1007/978-3-319-39519-7_7

[4] Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. DOI: https://doi.org/10.1016/0304-3975(87)90045-4

[5] Martin Hofmann. 2000. A Type System for Bounded Space and Functional In-Place Update. *Nord. J. Comput.* 7, 4 (2000), 258–289.

[6] Shams Imam and Vivek Sarkar. 2015. The Eureka Programming Model for Speculative Task Parallelism. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic.* 421–444. DOI: https://doi.org/10.4230/LIPIcs.ECOOP.2015.421

[7] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. 2005. *Java Concurrency in Practice.* Addison-Wesley Professional.

[8] Hari K. Pyla, Calvin J. Ribbens, and Srinidhi Varadarajan. 2011. Exploiting coarse-grain speculative parallelism. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 555–574. DOI: https://doi.org/10.1145/2048066.2048110