This is the accepted version of a paper presented at TyDe 2017, September 3, Oxford, UK.

Citation for the original published paper:

Affine killing: Semantics for stopping the ParT.

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-326193
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:
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:

```java
5 end
6 fun findFriend(user: User, t: Twitter, fb: Facebook): Par[Info]
7     val twInfo = (t) >> (fun x => x.findInfoTw(user))
8     val fbInfo = (fb) >> (fun x => x.findInfoFb(user))
9     twInfo >> updateDB
10    getPhoneNumber << (twInfo || fbInfo)
11 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] \(\rightarrow\) (t \(\rightarrow\) t) \(\rightarrow\) 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 (\(\ll\) :: Par[t] \(\rightarrow\) Par[t] \(\rightarrow\) Par[t]) produces a new ParT that groups the ParTs twInfo and fbInfo (line 11). Afterwards, the prune combinator (\(\ll\ll\) :: (Par[Maybe[t]]) \(\rightarrow\) 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:

```java
1 class Facebook
2    def findInfoFb(user: User): Info
3
4 end
5
6 fun findFriend(user: read User, t: read Twitter, fb: read Facebook): lin Par[Info]
7     . . .
8 end
9 . . .
```
This refined example adds affine annotations \( \text{lin} \) and \( \text{read} \) (lines 2, 7 and 8) to indicate whether the variables can be aliased, where \( \text{lin} \) does not allow aliasing but \( \text{read} \) 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, \( \text{lin} \) < \( \text{read} \), so that a linear Par\( T \) 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:

\[
\Delta; \Gamma_1 \vdash \text{twInfo} \leftrightarrow \text{twInfo}' : \kappa_1 \text{Par}[T] \\
\Delta; \Gamma_2 \vdash \text{fbInfo} \leftrightarrow \text{fbInfo}' : \kappa_2 \text{Par}[T] \\
\Delta; \Gamma_1 \Gamma_2 \vdash \text{twInfo} || \text{fbInfo} \leftrightarrow \text{twInfo}' || \text{fbInfo}' : \text{lin} \text{Par}[T]
\]

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

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

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

These annotations enable a kind of dynamic dispatch based on the structure of the Par\( T \). 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 \( [\text{fbInfo}]_{\text{lin}} \) and allocating new memory for the new Par\( T \) resulting from the other one (as \( [\text{twInfo}]_{\text{read}} \) 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 \( \text{twInfo} \) Par\( T \) has multiple dependencies (namely the update of the database and the possibility to fetch a phone number). These dependencies – statically caught by the affine mode – prevent the runtime from killing its underlying ongoing computations as that would be unsound. The case for the \( \text{fbInfo} \) Par\( T \) 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 (we have abbreviated \( \text{twInfo} \) to \( t \) and \( \text{fbInfo} \) to \( b \)):

\[\text{[Prune]}\]

\[
(t_{\exists} g E[v \ll \text{lin} \ (\{t_1\}_{\text{read}} || \text{lin} (b)_{\text{lin}})]) \rightarrow \\
(t_{\exists} (\text{peek}_{\text{lin}} \ (\{t_1\}_{\text{read}} || \text{lin} (b)_{\text{lin}})) (\text{fut}_{\exists} \ (t_{\exists} g E[v \ f]))
\]

\[\text{[Linear Peeking]}\]

\[
(t_{\exists} (\text{peek}_{\text{lin}} \ (\{t_1\}_{\text{read}} || \text{lin} (b)_{\text{lin}}))) \rightarrow \\
(t_{\exists} (\text{Just } t)) (\text{kill } g) \bigcup (\text{kill } h) \\
\text{h} \in \text{linDeps}(g)
\]

\[\text{[Kill]}\]

\[
(t_{\exists} g (\text{kill } g) \rightarrow (\text{kill } g) \bigcup (\text{kill } h) \\
\text{h} \in \text{linDeps}(g)
\]

The prune combinator relies on the hidden runtime function \( \text{peek} [3] \) that kills the speculative computations (rule Linear Peeking) applying thread-local reasoning: safely traversing asynchronous computations spawned by the \( \text{fbInfo} \) \( (\text{linDeps}(b)) \) 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 Par\( T \), 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 Par\( Ts \), 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 Par\( T \) 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


