Formally Proving a Compiler Transformation Safe

Joachim Breitner
Karlsruhe Institute of Technology
Germany
breitner@kit.edu

Abstract

We prove that the Call Arity analysis and transformation, as implemented in the Haskell compiler GHC, is *safe*, i.e. does not impede the performance of the program. We formalized syntax, semantics, the analysis and the transformation in the interactive theorem prover Isabelle to obtain a machine-checked proof and hence a level of rigor rarely obtained for compiler optimization safety theorems. The proof is modular and introduces *trace trees* as a suitable abstraction in abstract cardinality analyses. We discuss the breadth of the formalization gap.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

Keywords Functional programming, arity analysis, cardinality analysis, interactive theorem proving

1. Introduction

A lot of the fun in working on compilers, especially those that are actively used, lies in coming up with clever analyses and transformations that make the compiler produce better code. Such developments are regularly the topic of well-received academic publications. The emphasis in such papers tends to be on the empirical side – awesome benchmark results, elegant implementations, real-world impact.

A more formal, theoretical treatment is, however, not always given. Sometimes, a proof of functional correctness is included, which shows that the transformation will not change the *meaning* of the program. But for an optimization, we not only care about its functional correctness but also that the transformed program does not exhibit reduced performance. This operational property, here called *safety* (following [11]), is invisible to the semantics commonly employed in correctness proofs.

And even if a proof of safety is given, this is rarely performed in a machine-verified way, which would provide the highest level of assurance on the correctness of the proof.

Haskell'15, September 03-04 2015, Vancouver, BC, Canada.
Copyright © 2015 ACM 978-1-4503-3808-0/15/09…$15.00.
http://dx.doi.org/10.1145/2804302.2804312

In this work, we went all the way: We looked at the Call Arity analysis, formalized it in the interactive theorem prover Isabelle and created a machine-checked proof not only of functional correctness, but also that the performance of the transformed program is not worse than the original one's.

The purpose of an arity analysis is to detect when function definitions can be eta-expanded to take more arguments at once, which allows the compiler to generate more efficient code when calling such a function. Call Arity [5], which was recently added to GHC, combines an arity analysis with a novel cardinality analysis based on co-call graphs to gain more precision in the presence of recursion. This precision is required to effectively allow *foldl* and related combinators to take part in list fusion.

The cardinality analysis, which determines how often a function or a thunk is called, is required to eta-expand a thunk, as that is only safe if the thunk is called at most once. If the cardinality analysis were wrong and we would eta-expand a thunk that is called multiple times, we would lose the benefits of sharing and suddenly repeat work.

A correctness proof with regard to a standard denotational semantics would not rule that out! A more detailed semantics is required instead. We use an abstract machine with an explicit heap to prove that the number of heap allocations does not increase by transforming the program, and explain why this is a suitable criterion for safety.

Our contributions are:

- We provide a rigorous formal description of Call Arity and prove that it is safe, i.e. the transformed program does not perform more allocations than the original program.
- Our proof is modular. We cleanly separate the arity analysis part (Sec. 4) from the cardinality part, and divide the cardinality part into a three-stage refinement proof (Sec. 5). This gives greater insight into their interaction, and provides reusable components for similar proofs.
- We introduce infinite *trace trees* (Sec. 5.2) as a suitable domain for an abstract cardinality analysis.
- We formalized a suitable semantics akin to Sestoft's mark 1 abstract machine, the Call Arity analysis, the transformation and the safety proof in the theorem prover Isabelle. This gives us very high assurance of the correctness of this work, but also provides a data point on the question of how feasible machine-checked proofs of compiler transformations currently are (Sec. 6).
- Finally, and of more general interest, we critically discuss the formalization gap left by our formalization and find that the gap is not always bridgeable by meta-arguments. In particular, we explain how the interaction of Call Arity with other components of the compiler effected a serious and intricate bug, despite the proof of correctness (Sec. 6.1).

2. Overview and example

The remainder of the paper will necessarily be quite formal. In order to give a better intuition and overview, we first look at a small example in this section, and introduce the syntax, semantics, transformations and analyses more rigorously in the subsequent sections. A more elaborate motivation and explanation of the Call Arity analysis, including its effect on list fusion and benchmark results, can be found in [5].

2.1 From the example…

Consider the following Haskell program:

```
foo :: Int → Int
foo a = let t1, t2 :: Int → Int
            t1 = f1 a
            t2 = f2 a
        in  let g :: Int → Int → Int
                g x = if p x then t1 else g (x + t2 x)
            in  g 1 2
```

Here two thunks, *t1* and *t2*, are called from a recursive inner function *g*. They are thunks, because their definition is not in head normal form, so upon their first call, *f1* resp. *f2* will be called with the argument *a*, and the resulting value will be stored and used in later invocations of *t1* resp. *t2*.

As it stands, the function invocation *g 1 2* will be compiled to rather inefficient code: The caller will have to evaluate *g 1*, which creates and returns a function closure. This will be analyzed for the number of arguments it expects, and only then *2* will be pushed onto the stack and the closure will be entered [19]. If *g* would take two arguments directly, the call to *g* would simply push *1* and *2* onto the stack and execute the code for *g*, or even pass them in registers, which would be much faster.

The same reasoning applies to *t1* and *t2*. Generally, we want to eta-expand a definition to match the number of arguments it is called with.

We can actually eta-expand *g* to take two parameters: It is called with two arguments in the first place, and – assuming *g* is always called with two arguments – it calls itself with two arguments as well. So we may transform the definition of *g* to

$$g\ x\ y = (\textbf{if}\ p\ x\ \textbf{then}\ t1\ \textbf{else}\ g\ (x + t2\ x))\ y,$$

which would then be further simplified by the compiler to

$$g\ x\ y = \textbf{if}\ p\ x\ \textbf{then}\ t1\ y\ \textbf{else}\ g\ (x + t2\ x)\ y.$$

We now see that both *t1* and *t2* are always called with one argument. Can we eta-expand their definitions to *t1* y = *f1 a y* resp. *t2* y = *f2 a y*? It depends!

If we eta-expand *t2* then the evaluation of *f2 a* will no longer be shared between multiple invocations of *t2*. As we do not know anything about *f2* we have to pessimistically assume this to be an expensive operation that we must not suddenly repeat. We expect *t2* to be called multiple times here, so a conservative arity analysis must not eta-expand it.

For *t1*, we can do better: It is definitely called at most once, so it is safe to eta-expand its definition. That is why a good arity analysis needs the help of a precise cardinality analysis. How would that analysis figure that out? The body of *g* on its own calls both *t2* and *t1* at most once, so having cardinality information for subexpressions is not enough to attain such precision, and our cardinality analysis needs to keep track of more.

The Call Arity analysis comes with a cardinality analysis based on the notion of *co-call graphs*. In these (non-transitive) graphs edges connect variables that might be called together.

Looking at the definition of *g*, we see that *p* is called together with all the other variables, and *g* is called together with *t2*. Thus, the resulting graph is $\begin{tikzcd} t1 \arrow[r, dash] & p \arrow[r, dash] \arrow[rr, bend left, dash] & t2 \arrow[r, dash] & g \end{tikzcd}$. In particular, we can see that *g* and *t1* are not going to be called together.

Together with the fact that the body of the **let**-binding calls *g* at most once, we can describe the calls originating from the inner **let** with the graph $\begin{tikzcd} t1 \arrow[r, dash] \arrow[rr, bend left, dash] & p \arrow[r, dash] \arrow[loop below, dash] & t2 \arrow[loop below, dash] \end{tikzcd}$: Both *g* and *t2* can be called multiple times, but the absence of a loop at *t1* implies the desired cardinality information: *t1* is called at most once.

2.2 …to the general case

This explanation might have been convincing for this example, but how would we prove that the analysis and transformation are safe in general?

In order to do so, we first need a suitable semantics. The elegant standard denotational semantics for functional programs are unfortunately too abstract and admit no observation of program performance. Therefore, we use a standard small-step operational semantics similar to Sestoft's mark 1 abstract machine. It defines a relation $(\Gamma, e, S) \Rightarrow^* (\Gamma', e', S')$ between configurations consisting of a heap, a control, i.e. the current expression under evaluation, and a stack (Sec. 3).

With that semantics, we could follow Sands [21] and measure performance by counting evaluation steps. But that is too finegrained: Our eta-expansion transformation causes additional beta-reductions to be performed during evaluation, and without subsequent simplification – which does happen in a real compiler, but which we do not want to include in the proof – these increase the number of steps in our semantics.

Therefore, we measure the performance by counting the number of allocations performed during the evaluation. This is sufficient to detect accidental duplication of work, as shown by this gedankenexperiment: Consider a program e_1, which is transformed to e_2, and a subexpression e of e_1 that also occurs in e_2. By replacing e with **let** *x1* = *x1*,…, *xn* = *xn* **in** *e*, where the variables are fresh, we can force each evaluation of e to perform at least n allocations, for an arbitrary large choice of n. So unless e_2 evaluates e at most as often as e_1 does, we can choose n large enough to make e_2 allocate more than e_1. Conversely, if our criterion holds, we can conclude that the transformation does not duplicate work.

This measure is also realistic: When working on GHC, the number of bytes allocated by a benchmark or a test case is the prime measure that developers observe to detect improvements and regressions, as in practice, it correlates very well with execution time and memory usage, while being more stable across differing environments.

A transformation is *safe* in this sense if the transformed program performs no more allocations than the original program.

The arity transformation eta-expands expressions, so in order to prove it safe, we need to identify conditions when eta-expansion itself is safe, and ensure that these conditions are always met.

A sufficient condition for the safety of an n-fold eta-expansion of an expression e is that whenever e is evaluated, the top n elements on the stack are arguments, as stated in Lemma 1. The safety proof for the arity analysis (Lemma 2) keeps track of some invariants during the evaluation which ensure that we can apply Lemma 1 whenever an eta-expanded expression is about to be evaluated.

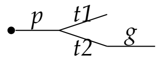
The proof is first performed for a naive arity analysis without a cardinality analysis, before formally introducing the concept of a cardinality analysis in Sec. 5. We do not simply prove safety of the co-call graph based analysis directly, but split it up into a series of increasingly concrete proofs, each building on the result of the previous, for two reasons:

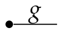
- It is nice to separate various aspects of the proof (i.e. the interaction of the arity analysis with the cardinality analysis; the gap between the steps of the semantics and the structurally recursive nature of the analysis; different treatments of recursive and non-recursive bindings) into individual steps, but more importantly
- while the co-call graph data structure is sufficiently expressive to implement the analysis, it is an unsuitable abstraction for the safety proof, as it cannot describe the recursion patterns of a heap, where some expressions are calling each other in a nice, linear fashion among other, more complex recursion patterns.

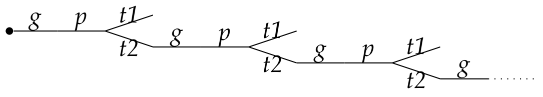
In the first refinement, the cardinality analysis is completely abstract: Its input is the whole configuration and its result is simply which variables on the heap are going to be called more than once. In our example, after *t1*, *t2* and g have been put on the heap, this analysis would find out that *t2* and g are called more than once, but not *t1*. We give conditions (Definition 6) when an arity analysis using such a cardinality analysis is safe (Lemma 3).

The next refinement assumes a cardinality analysis that now looks just at expressions, not whole configurations, and returns a much richer analysis result: A *trace tree*, which is a (possibly) infinite tree where each path corresponds to one possible execution and the edges are labeled by the variables called during that evaluation.

In our example, the tree corresponding to the right-hand-side of g, namely



can be combined with the very simple tree  from the body of the inner **let** to form the infinite tree



which describes the overall sequence of calls. Clearly, on every possible path, *t1* is called at most once.

Given such a trace tree analysis, an abstract analysis as described in the first refinement can be implemented: The trees describing the expressions in a configuration (on the heap, as the control or in the stack) can be combined to a tree describing the behavior of the whole configuration. This calculation, named s in Sec. 5.2, is quite natural for trace trees, but would be hard to define on co-call graphs only. From that tree we can determine the cardinalities of the individual variables. We specify conditions on the trace tree analysis (Definition 9) and in Lemma 4 show them to be sufficient to fulfill the specification of the first refinement.

The third and final refinement assumes an analysis that returns a co-call graph for each expression. Co-call graphs can be seen as compact approximations of trace trees, with edges between variables that can occur on the same path in the tree. The specification in Definition 10 is shown in Lemma 5to be sufficient to fulfill the specification of the second refinement.

Eventually, we give the definition of the real Call Arity analysis in Sec. 5.4, and as it fulfills the specification of the final refinement, the desired safety theorem (Theorem 1) follows.

The following three technical sections necessarily omit some detail, especially in the proofs. But since the machine-checked formalization exists, such omissions needn't cause worry. The full Isabelle code is available at [4]; the proof document contains a table that maps the definitions and lemmas of this paper to the corresponding entities in the Isabelle development.

$x, y, z\colon$	Var	variables
$e, v\colon$	Expr	expressions
$e ::=$	x	variable
	$\mid e\ x$	application
	$\mid \lambda x.\, e$	lambda abstraction
	$\mid \mathsf{C_t} \mid \mathsf{C_f}$	constructor
	$\mid e\ \mathsf{?}\ e_\mathsf{t} : e_\mathsf{f}$	case analysis
	$\mid \mathsf{let}\ \Gamma\ \mathsf{in}\ e$	mutually recursive bindings
$\Gamma, \Delta\colon$	$\text{Var} \rightharpoonup \text{Expr}$	heaps, bindings

Figure 1. A simple lambda calculus

3. Syntax and semantics

Call Arity operates on GHC's intermediate language Core, but that is too large for our purposes: The analysis completely ignores types, so we would like to work on an untyped representation.

Additionally, we do not need the full expressiveness of algebraic data types. We use booleans ($\mathsf{C_t}$, $\mathsf{C_f}$) with an **if-then-else** construct as representatives for case analysis on data types. The "other" interesting feature of data constructors, i.e. that they are values that can contain possibly unevaluated code, can already be observed with function closures.

Our syntax is given in Figure 1. The bindings of a **let** are represented as finite maps from variables to expressions; the same type is used to describe a heap.

Like Launchbury [16] and others [12, 13, 25], we require application arguments to be variables. This ensures that all bindings on the heap are created by a **let** and we do not have to ensure separately that the evaluation of a function's argument is shared.

We denote the set of free variables of an expression e (or another object containing expressions) with $\mathsf{fv}(e)$, and $e[x := y]$ is the expression e with every free occurrence of the variable x replaced by y. The predicate $\mathsf{isVal}(e)$ holds iff e is a lambda abstraction or a constructor.

A heap Γ is a partial map from variable names to expressions. The set $\mathsf{dom}\,\Gamma := \{x \mid (x \mapsto e) \in \Gamma\}$ contains all names bound in Γ, while $\mathsf{thunks}\,\Gamma := \{x \mid (x \mapsto e) \in \Gamma, \neg\,\mathsf{isVal}(e)\}$ contains just those that are bound to thunks. Note that we consider heap-bound names to be free, i.e. $\mathsf{dom}\,\Gamma \subseteq \mathsf{fv}\,\Gamma$.

The proper treatment of names is the major technical hurdle when rigorously formalizing anything related to the lambda calculus. We employ Nominal Logic [27] here, so the lambda abstractions and **let**-bindings are proper equivalency classes, i.e. $\lambda x.\, x = \lambda y.\, y$.

A *configuration* (Γ, e, S) consists of the heap Γ, the control e and the stack S. The stack is constructed from

- the empty stack, $[]$,
- arguments, written $\$x{\cdot}S$ and put on the stack during the evaluation of an application,
- update markers, written $\#x{\cdot}S$ and put on the stack during the evaluation of a variable's right-hand-side, and

$$
\begin{array}{lrl}
 & (\Gamma, e\ x, S) \Rightarrow (\Gamma, e, \$x\cdot S) & \text{APP}_1 \\
 & (\Gamma, \lambda y.\, e, \$x\cdot S) \Rightarrow (\Gamma, e[y := x], S) & \text{APP}_2 \\
(x \mapsto e) \in \Gamma \implies & (\Gamma, x, S) \Rightarrow (\Gamma \setminus x, e, \#x\cdot S) & \text{VAR}_1 \\
\mathsf{isVal}(e) \implies & (\Gamma, e, \#x\cdot S) \Rightarrow (\Gamma[x \mapsto e], e, S) & \text{VAR}_2 \\
 & (\Gamma, (e \mathbin{?} e_{\mathbf{t}} : e_{\mathbf{f}}), S) \Rightarrow (\Gamma, e, (e_{\mathbf{t}} : e_{\mathbf{f}})\cdot S) & \text{IF}_1 \\
b \in \{\mathbf{t}, \mathbf{f}\} \implies & (\Gamma, \mathsf{C}_b, (e_{\mathbf{t}} : e_{\mathbf{f}})\cdot S) \Rightarrow (\Gamma, e_b, S) & \text{IF}_2 \\
\mathsf{dom}\,\Delta \cap \mathsf{fv}(\Gamma, S) = \{\} \implies & & \\
 & (\Gamma, \mathbf{let}\ \Delta\ \mathbf{in}\ e, S) \Rightarrow (\Delta \cdot \Gamma, e, S) & \text{LET}
\end{array}
$$

Figure 2. The operational semantics

- alternatives, written $(e_1 : e_2)\cdot S$ and put on the stack during the evaluation of the scrutinee of an **if-then-else** construct.

Throughout this work we assume all configurations to be *good*, i.e. $\mathsf{dom}\,\Gamma$ and $\#S := \{x \mid \#x \in S\}$ are disjoint and the update markers on the stack are distinct.

Following Sestoft [25], we define the semantics via the single step relation $\Rightarrow$, defined in Figure 2. We write $\Rightarrow^*$ for the reflexive transitive closure of this relation, which describes a particular *execution*.

In the interest of naming hygiene, the names for the new bindings in the LET rule have to be fresh with regard to what is already present on the heap and stack, as ensured by the side-condition. An interesting side-effect is that this rule, and hence the whole semantics, is not deterministic, as there is an infinite number of valid names that can be used when putting the bindings onto the heap. Consequently, our proofs cannot take short-cuts using determinism, which would be a problem if "real" nondeterminism were added to the formalism.

Note that the semantics takes good configurations to good configurations.

This semantics is equivalent to Launchbury's natural semantics [16], which in turn is correct and adequate with regard to a standard denotational semantics; these proofs are machine-verified as well [3].

3.1 Arities and Eta-Expansion

Eta-expansion replaces an expression e by $(\lambda x.\, e\ x)$. The n-fold eta-expansion is described by $\mathcal{E}_n(e) := (\lambda z_1 \ldots z_n.\, e\ z_1 \ldots\ z_n)$, where the z_i are distinct and fresh with regard to e. We thus consider an expression e to have *arity* $\alpha \in \mathbb{N}$ if we can replace it by $\mathcal{E}_\alpha(e)$ without negative effect on the performance.

Other analyses determine the arity based on the definition of e, i.e. its *internal* arity [28]. Here, we treat e as a black box and instead look at its context to determine its *external* arity. For that, we can give an alternative definition: An expression e has arity α if upon every evaluation of e, there are at least α arguments on the stack.

If an expression has arity α, then it also has arity α' for $\alpha' \leq \alpha$; every expression has arity 0. Our lattice therefore is:

$$\cdots \sqsubset 3 \sqsubset 2 \sqsubset 1 \sqsubset 0.$$

For convenience, we set $0 - 1 = 0$. By convention, $\bar{\alpha}$ is a partial map from variable names to arities, and $\dot{\alpha}$ is a list of arities.

4. Arity analyses

An arity analysis is thus a function that, given a binding (Γ, e), consisting of variable names bound to right-hand-sides in Γ and the body e, determines the arity of each of the bound expressions. It depends on the number α of arguments passed to e and may return $\bot$ for a name that is not called at all:

$$\mathcal{A}_\alpha(\Gamma, e) \colon \mathsf{Var} \to \mathbb{N}_\bot.$$

Given such an analysis, we can run it over a program and transform it accordingly. We traverse the syntax tree, while keeping track of the number of arguments passed:

$$
\begin{aligned}
\mathsf{T}_\alpha(x) &= x \\
\mathsf{T}_\alpha(e\ x) &= \mathsf{T}_{\alpha+1}(e)\ x \\
\mathsf{T}_\alpha(\lambda x.\, e) &= (\lambda x.\, \mathsf{T}_{\alpha-1}(e)) \\
\mathsf{T}_\alpha(\mathsf{C}_b) &= \mathsf{C}_b \qquad \text{for } b \in \{\mathbf{t}, \mathbf{f}\} \\
\mathsf{T}_\alpha(e \mathbin{?} e_{\mathbf{t}} : e_{\mathbf{f}}) &= \mathsf{T}_0(e) \mathbin{?} \mathsf{T}_\alpha(e_{\mathbf{t}}) : \mathsf{T}_\alpha(e_{\mathbf{f}}) \\
\mathsf{T}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e) &= \mathbf{let}\ \overline{\mathsf{T}}_{\mathcal{A}_\alpha(\Gamma,e)}(\Gamma)\ \mathbf{in}\ \mathsf{T}_\alpha(e)
\end{aligned}
$$

The actual transformation happens at a binding, where we eta-expand bound expressions according to the result of the arity analysis. If the analysis determines that a binding is never called, we simply leave it alone:

$$\overline{\mathsf{T}}_{\bar{\alpha}}(\Gamma) = \left[x \mapsto \begin{cases} e & \text{if } \bar{\alpha}(x) = \bot \\ \mathcal{E}_\alpha(\mathsf{T}_\alpha(e)) & \text{if } \bar{\alpha}(x) = \alpha \end{cases} \middle| (x \mapsto e) \in \Gamma\right].$$

As motivated earlier, we consider an arity analysis $\mathcal{A}$ safe if the transformed program does not perform more allocations than the original program. A – technical – benefit of this measure is that the number of allocations always equals the size of the heap plus the number of update markers on the stack, as no garbage collector is modeled in our semantics:

Definition 1 (Safe transformation) A program transformation T is *safe* if for every execution

$$([], e, []) \Rightarrow^* (\Gamma, v, [])$$

with $\mathsf{isVal}(v)$, there is an execution

$$([], \mathsf{T}(e), []) \Rightarrow^* (\Gamma', v', [])$$

with $\mathsf{isVal}(v')$ and $|\,\mathsf{dom}\,\Gamma'| \leq |\,\mathsf{dom}\,\Gamma|$.

An arity analysis $\mathcal{A}$ is safe if the transformation T is safe. □

Specification We begin by stating sufficient conditions for an arity analysis to be safe. In order to phrase the conditions, we also need to know the arities an expression e calls its free variables with, assuming it is itself called with α arguments:

$$\mathcal{A}_\alpha(e) \colon \mathsf{Var} \to \mathbb{N}_\bot$$

For notational simplicity, we define $\mathcal{A}_\bot(e) := \bot$.

The specification consists of a few naming hygiene conditions and an inequality for most syntactical constructs:

Definition 2 (Arity analysis specification)

$$
\begin{array}{lr}
\mathsf{dom}\,\mathcal{A}_\alpha(e) \subseteq \mathsf{fv}\,e & \text{(A-dom)} \\
\mathsf{dom}\,\mathcal{A}_\alpha(\Gamma, e) \subseteq \mathsf{dom}\,\Gamma & \text{(Ah-dom)} \\
z \notin \{x, y\} \implies \mathcal{A}_\alpha(e[x := y])\ z = \mathcal{A}_\alpha(e)\ z & \text{(A-subst)} \\
x, y \notin \mathsf{dom}\,\Gamma \implies & \\
\mathcal{A}_\alpha(\Gamma[x := y], e[x := y]) = \mathcal{A}_\alpha(\Gamma, e) & \text{(Ah-subst)} \\
[x \mapsto \alpha] \sqsubseteq \mathcal{A}_\alpha(x) & \text{(A-Var)} \\
\mathcal{A}_{\alpha+1}(e) \sqcup [x \mapsto 0] \sqsubseteq \mathcal{A}_\alpha(e\ x) & \text{(A-App)} \\
\mathcal{A}_{\alpha-1}(e) \setminus \{x\} \sqsubseteq \mathcal{A}_\alpha(\lambda x.\, e) & \text{(A-Lam)} \\
\mathcal{A}_0(e) \sqcup \mathcal{A}_\alpha(e_{\mathbf{t}}) \sqcup \mathcal{A}_\alpha(e_{\mathbf{f}}) \sqsubseteq \mathcal{A}_\alpha(e \mathbin{?} e_{\mathbf{t}} : e_{\mathbf{f}}) & \text{(A-If)} \\
\overline{\mathcal{A}}_{\mathcal{A}_\alpha(\Gamma,e)}(\Gamma) \sqcup \mathcal{A}_\alpha(e) \sqsubseteq \mathcal{A}_\alpha(\Gamma, e) \sqcup \mathcal{A}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e) & \\
 & \text{(A-Let)}
\end{array}
$$

where $\overline{\mathcal{A}}_{\bar{\alpha}}(\Gamma) := \bigsqcup \{\mathcal{A}_{(\bar{\alpha}\ x)}(e) \mid (x \mapsto e) \in \Gamma\}$. □

These conditions come quite naturally: An expression should not report calls to names that it does not know about. Replacing one variable by another should not affect the arity of other variables. A variable, evaluated with a certain arity, should report (at most) that arity.

In the rules for application and lambda abstraction we keep track of the number of arguments. As we model a forward analysis which looks at bodies before right-hand-sides, we get no useful information on how the argument x in an application $e\ x$ is called by e.

In rule (A-If), the scrutinee is evaluated without arguments, hence it is analyzed with arity 0.

The rule (A-Let) is a concise way to capture a few requirements. Note that, by (A-dom) and (Ah-dom), the domains of $\mathcal{A}_\alpha(\Gamma, e)$ and $\mathcal{A}_\alpha(\textbf{let}\ \Gamma\ \textbf{in}\ e)$ are disjoint, i.e. $\mathcal{A}_\alpha(\Gamma, e)$ contains the information on how the names of the current binding are called, while $\mathcal{A}_\alpha(\textbf{let}\ \Gamma\ \textbf{in}\ e)$ informs us about the free variables. The left-hand side contains all possible calls, both from the body of the binding and from each bound expression. These are analyzed with the arity reported by $\mathcal{A}_\alpha(\Gamma, e)$. The occurrence of $\mathcal{A}_\alpha(\Gamma, e)$ on both sides of the inequality anticipates the fixed-point iteration in the implementation of the analysis.

Definition 2 suffices to prove functional correctness, i.e.

$$[\![\mathsf{T}_0(e)]\!] = [\![e]\!],$$

holds, but not safety, as the issue of thunks is not touched upon yet. Without the aid of a cardinality analysis, an arity analysis has to simply give up when it comes across a thunk:

Definition 3 (No-cardinality analysis specification)

$$x \in \mathsf{thunks}\,\Gamma \implies \mathcal{A}_\alpha(\Gamma, e)\ x = 0 \qquad \text{(Ah-thunk)}$$

□

Safety The safety of an eta-expanding transformation rests on the simple observation that, given enough arguments on the stack, an eta-expanded expression evaluates to the original expression:

Lemma 1 $(\Gamma, \mathcal{E}_\alpha(e), \$x_1 \cdots \$x_\alpha \cdot S) \Rightarrow^* (\Gamma, e, \$x_1 \cdots \$x_\alpha \cdot S)$ □

PROOF By APP_2, invoked α times, followed by APP_1, α times. ■

So the safety proof for the whole transformation now just has to make sure that whenever we evaluate an eta-expanded value, there are enough arguments on top of the stack. Let $\mathsf{args}(S)$ denote the number of arguments on top of the stack.

During evaluation, we need to construct the transformed configurations. Therefore, we need to keep track of the arity argument to each contained expressions: those on the heap, the control and those in alternatives on the stack. Together, these arguments form an *arity annotation* written $(\bar{\alpha}, \alpha, \dot{\alpha})$. Given such an annotation, we can transform a configuration:

$$\mathsf{T}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S)) = (\overline{\mathsf{T}}_{\bar{\alpha}}(\Gamma), \mathsf{T}_\alpha(e), \dot{\mathsf{T}}_{\dot{\alpha}}(S))$$

where the stack is transformed by

$$\begin{aligned}
\dot{\mathsf{T}}_{\alpha\cdot\dot{\alpha}}((e_\mathbf{t} : e_\mathbf{f})\cdot S) &= (\mathsf{T}_\alpha(e_\mathbf{t}) : \mathsf{T}_\alpha(e_\mathbf{f}))\cdot\dot{\mathsf{T}}_{\dot{\alpha}}(S)\\
\dot{\mathsf{T}}_{\dot{\alpha}}(\$x\cdot S) &= \$x\cdot\dot{\mathsf{T}}_{\dot{\alpha}}(S)\\
\dot{\mathsf{T}}_{\dot{\alpha}}(\#x\cdot S) &= \#x\cdot\dot{\mathsf{T}}_{\dot{\alpha}}(S)\\
\dot{\mathsf{T}}_{\dot{\alpha}}([]) &= [].
\end{aligned}$$

While carrying the arity annotation through the evaluation of our programs, we need to ensure that it stays *consistent* with the current configuration.

Definition 4 (Arity annotation consistency) An arity annotation is consistent with a configuration, written $(\bar{\alpha}, \alpha, \dot{\alpha}) \triangleright (\Gamma, e, S)$, if

- $\mathsf{dom}\,\bar{\alpha} \subseteq \mathsf{dom}\,\Gamma \cup \#S$,
- $\mathsf{args}(S) \sqsubseteq \alpha$,
- $\left(\overline{\mathcal{A}}_{\bar{\alpha}}(\Gamma) \sqcup \mathcal{A}_\alpha(e) \sqcup \dot{\mathcal{A}}_{\dot{\alpha}}(S)\right)\big|_{\mathsf{dom}\,\Gamma\cup\#S} \sqsubseteq \bar{\alpha}$, where

$$\begin{aligned}
\dot{\mathcal{A}}_{[]}([]) &:= \bot\\
\dot{\mathcal{A}}_{\alpha\cdot\dot{\alpha}}((e_\mathbf{t} : e_\mathbf{f})\cdot S) &:= \mathcal{A}_\alpha(e_\mathbf{t}) \sqcup \mathcal{A}_\alpha(e_\mathbf{f}) \sqcup \dot{\mathcal{A}}_{\dot{\alpha}}(S)\\
\dot{\mathcal{A}}_{\dot{\alpha}}(\$x\cdot S) &:= [x \mapsto 0] \sqcup \dot{\mathcal{A}}_{\dot{\alpha}}(S)\\
\dot{\mathcal{A}}_{\dot{\alpha}}(\#x\cdot S) &:= [x \mapsto 0] \sqcup \dot{\mathcal{A}}_{\dot{\alpha}}(S), \text{ and}
\end{aligned}$$

- $\dot{\alpha} \triangleright S$, defined as

$$\begin{aligned}
&[] \triangleright []\\
&\alpha\cdot\dot{\alpha} \triangleright (e_\mathbf{t} : e_\mathbf{f})\cdot S \iff \dot{\alpha} \triangleright S \wedge \mathsf{args}(S) \sqsubseteq \alpha\\
&\dot{\alpha} \triangleright \$x\cdot S \iff \dot{\alpha} \triangleright S\\
&\dot{\alpha} \triangleright \#x\cdot S \iff \dot{\alpha} \triangleright S.
\end{aligned}$$

□

As this definition does not consider the issue of thunks, we extend it to

Definition 5 (No-cardinality arity annotation consistency) defined as $(\bar{\alpha}, \alpha, \dot{\alpha}) \triangleright_\mathbf{N} (\Gamma, e, S)$, iff $(\bar{\alpha}, \alpha, \dot{\alpha}) \triangleright (\Gamma, e, S)$ and $\bar{\alpha}\ x = 0$ for all $x \in \mathsf{thunks}\,\Gamma$. □

We do not include this requirement in definition of $\triangleright$ as we extend it differently when we add a cardinality analysis.

Clearly $(\bot, 0, [])$ is a consistent annotation for an initial configuration $([], e, [])$. We will take consistently annotated configurations to consistently annotated configurations during the evaluation – with one exception, which causes a minor technical overhead: Upon evaluation of a variable x, its binding $x \mapsto e$ is always taken off the heap first, even when it is already evaluated, i.e. $\mathsf{isVal}(e)$:

$$(\Gamma[x \mapsto e], x, S) \Rightarrow (\Gamma, e, \#x\cdot S) \Rightarrow (\Gamma[x \mapsto e], e, S)$$

We would not be able to prove consistency in the intermediate state. To work around this issue, assume that rule VAR_1 has an additional constraint $\neg\,\mathsf{isVal}(e)$ and that the rule

$$(x \mapsto e) \in \Gamma,\ \mathsf{isVal}(e) \implies (\Gamma, x, S) \Rightarrow (\Gamma, e, S) \quad (\text{VAR}_1')$$

is added. This modification makes the semantics skip over one step, which is fine (and closer to what happens in reality).

Lemma 2 *Assume $\mathcal{A}$ fulfills the Definitions 2 and 3.*

If we have $(\Gamma, e, S) \Rightarrow^* (\Gamma', e', S')$ and $(\bar{\alpha}, \alpha, \dot{\alpha}) \triangleright_\mathbf{N} (\Gamma, e, S)$, then there exists an arity annotation $(\bar{\alpha}', \alpha', \dot{\alpha}')$ with $(\bar{\alpha}', \alpha', \dot{\alpha}') \triangleright_\mathbf{N} (\Gamma', e', S')$, and $\mathsf{T}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S)) \Rightarrow^* \mathsf{T}_{(\bar{\alpha}',\alpha',\dot{\alpha}')}((\Gamma', e', S'))$. □

PROOF by the individual steps of $\Rightarrow^*$. For APP_1 we have

$$\begin{aligned}
\mathcal{A}_{\alpha+1}(e) \sqcup \dot{\mathcal{A}}_{\dot{\alpha}}(\$x\cdot S) &= \mathcal{A}_{\alpha+1}(e) \sqcup [x \mapsto 0] \sqcup \dot{\mathcal{A}}_{\dot{\alpha}}(S)\\
&\sqsubseteq \mathcal{A}_\alpha(e\ x) \sqcup \dot{\mathcal{A}}_{\dot{\alpha}}(S)
\end{aligned}$$

using (A-App) and the definition of $\dot{\mathcal{A}}$. So with $(\bar{\alpha}, \alpha, \dot{\alpha}) \triangleright_\mathbf{N} (\Gamma, e\ x, S)$ we have $(\bar{\alpha}, \alpha + 1, \dot{\alpha}) \triangleright_\mathbf{N} (\Gamma, e, \$x\cdot S)$. Furthermore

$$\begin{aligned}
\mathsf{T}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e\ x, S)) &= (\overline{\mathsf{T}}_{\bar{\alpha}}(\Gamma), (\mathsf{T}_{\alpha+1}(e))\ x, \dot{\mathsf{T}}_{\dot{\alpha}}(S))\\
&\Rightarrow (\overline{\mathsf{T}}_{\bar{\alpha}}(\Gamma), \mathsf{T}_{\alpha+1}(e), \$x\cdot\dot{\mathsf{T}}_{\dot{\alpha}}(S))\\
&= \mathsf{T}_{(\bar{\alpha},\alpha+1,\dot{\alpha})}((\Gamma, e, \$x\cdot S))
\end{aligned}$$

by rule APP_1.

The other cases follow this pattern, where the inequalities in Definition 2 ensure the preservation of consistency.

In case VAR$_1$ the variable x is bound to a thunk. From consistency we obtain $\bar{\alpha}\ x = 0$, so we can use $\mathcal{E}_0(\mathsf{T}_0(e)) = \mathsf{T}_0(e)$. Similarly, $\alpha = \bar{\alpha}\ x = 0$ holds in case VAR$_2$.

The actual eta-expansion is handled is case VAR$'_1$: We have

$$\mathsf{args}(\dot{\mathsf{T}}_{\dot{\alpha}}(S)) = \mathsf{args}(S) \sqsubseteq \alpha \sqsubseteq \mathcal{A}_\alpha(x)\ x \sqsubseteq \bar{\alpha}\ x,$$

from consistency and (A-Var) and hence

$$\begin{aligned}\mathsf{T}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, x, S)) &\Rightarrow (\overline{\mathsf{T}}_{\bar{\alpha}}(\Gamma), \mathcal{E}_{\bar{\alpha}\ x}(\mathsf{T}_{\bar{\alpha}\ x}(e)), \dot{\mathsf{T}}_{\dot{\alpha}}(S)) && \{\ \text{VAR}'_1\ \}\\ &\Rightarrow^* (\overline{\mathsf{T}}_{\bar{\alpha}}(\Gamma), \mathsf{T}_{\bar{\alpha}\ x}(e), \dot{\mathsf{T}}_{\dot{\alpha}}(S)) && \{\ \text{by Lemma 1}\ \}\\ &= \mathsf{T}_{(\bar{\alpha},\bar{\alpha}\ x,\dot{\alpha})}((\Gamma, e, S)). && \blacksquare\end{aligned}$$

The main take-away of this lemma is the following corollary, which states that the transformed program performs the same number of allocations as the original program.

Corollary 1 *The arity analysis is safe: If* $([], e, []) \Rightarrow^* (\Gamma, v, [])$, *then there exists* Γ' *and* v' *such that* $([], \mathsf{T}_0(e), []) \Rightarrow^* (\Gamma', v', [])$ *where* Γ *and* Γ' *contain the same number of bindings.* □

PROOF We have $(\bot, 0, []) \rhd_{\mathsf{N}} ([], e, [])$. Lemma 2 gives us $\bar{\alpha}$, α and $\dot{\alpha}$ so that $\mathsf{T}_{(\bot,0,[])}(([], e, [])) \Rightarrow^* \mathsf{T}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, v, []))$ and $\overline{\mathsf{T}}_{\bar{\alpha}}(\Gamma)$ binds the same names as Γ. ■

4.1 A concrete arity analysis

So far, we have a specification for an arity analysis and a proof that every analysis that fulfills the specification is safe.

One possible implementation is the trivial arity analysis, which does not do anything useful and simply returns the most pessimistic result: $\mathcal{A}_\alpha(e) := [x \mapsto 0 \mid x \in \mathsf{fv}\,e]$ and $\mathcal{A}_\alpha(\Gamma, e) := [x \mapsto 0 \mid x \in \mathsf{dom}\,\Gamma]$.

A more realistic arity analysis is defined by

$$\begin{aligned}\mathcal{A}_\alpha(x) &:= [x \mapsto \alpha]\\ \mathcal{A}_\alpha(e\ x) &:= \mathcal{A}_{\alpha+1}(e) \sqcup [x \mapsto 0]\\ \mathcal{A}_\alpha(\lambda x.\ e) &:= \mathcal{A}_{\alpha-1}(e) \setminus \{x\}\\ \mathcal{A}_\alpha(e\ \mathbf{?}\ e_{\mathbf{t}} : e_{\mathbf{f}}) &:= \mathcal{A}_0(e) \sqcup \mathcal{A}_\alpha(e_{\mathbf{t}}) \sqcup \mathcal{A}_\alpha(e_{\mathbf{f}})\\ \mathcal{A}_\alpha(\mathbf{C}_b) &:= \bot \qquad \text{for } b \in \{\mathbf{t}, \mathbf{f}\}\\ \mathcal{A}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e) &:=\end{aligned}$$

$$(\mu\bar{\alpha}.\ \overline{\mathcal{A}}_{\bar{\alpha}}(\Gamma) \sqcup \mathcal{A}_\alpha(e) \sqcup [x \mapsto 0 \mid x \in \mathsf{thunks}\,\Gamma]) \setminus \mathsf{dom}\,\Gamma$$

and

$$\mathcal{A}_\alpha(\Gamma, e) := (\mu\bar{\alpha}.\ \overline{\mathcal{A}}_{\bar{\alpha}}(\Gamma) \sqcup \mathcal{A}_\alpha(e) \sqcup [x \mapsto 0 \mid x \in \mathsf{thunks}\,\Gamma])\big|_{\mathsf{dom}\,\Gamma}$$

where $(\mu\bar{\alpha}.\ldots)$ denotes the least fixed point, which exists as the involved operations are continuous and monotone in $\bar{\alpha}$. Moreover, the fixed point can be found in a finite number of steps by iterating from $\bot$, as the carrier of $\bar{\alpha}$ is bounded by the finite set $\mathsf{fv}\,\Gamma \cup \mathsf{fv}\,e$, and the pointwise partial order on arities has no infinite ascending chains. As this ignores the issues of thunks, it corresponds to the analysis described by Gill [10].

This implementation fulfills Definition 2 and Definition 3, so by Corollary 1, it is safe.

5. Cardinality analyses

The previous section proved the safety of a straight-forward arity analysis. But it was severely limited by not being able to eta-expand thunks, which is desirable in practice.

5.1 Abstract cardinality analysis

So the arity analysis needs an accompanying *cardinality analysis* which prognoses how often a bound variable is going to be evaluated: This is modeled as a function

$$\mathcal{C}_\alpha(\Gamma, e) \colon \mathsf{Var} \to \mathsf{Card}$$

where Card is the three element lattice

$$\bot \sqsubset \mathbf{1} \sqsubset \infty,$$

corresponding to "not called", "called at most once" and "no information", respectively. We use γ for an element of Card and $\bar{\gamma}$ for a mapping $\mathsf{Var} \to \mathsf{Card}$.

The expression $\bar{\gamma} - x$, which subtracts one call from the prognosis, is defined as

$$(\bar{\gamma} - x)\ y = \begin{cases}\bot & \text{if } y = x \text{ and } \bar{\gamma}\ y = \mathbf{1}\\ \bar{\gamma}\ y & \text{otherwise.}\end{cases}$$

Specification We start with a very abstract specification for a safe cardinality analysis and prove that an arity transformation using it is still safe. We stay oblivious in how the analysis works and defer that to the next refinement step in Section 5.2.

For the specification we not only need the local view on one binding, as provided by $\mathcal{C}_\alpha(\Gamma, e)$, but also a prognosis on how often each variable is called by a complete and arity-annotated configuration:

$$\mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S)) \colon \mathsf{Var} \to \mathsf{Card}$$

Definition 6 (Cardinality analysis specification) The cardinality prognosis and cardinality analysis fulfill some obvious naming hygiene conditions:

$$\mathsf{dom}\,\mathcal{C}_\alpha(\Delta, e) = \mathsf{dom}\,\mathcal{A}_\alpha(\Delta, e) \qquad \text{(Ch-dom)}$$

$$\mathsf{dom}\,\mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S)) \subseteq \mathsf{fv}(\Gamma, e, S) \qquad \text{(C-dom)}$$

$$\bar{\alpha}\big|_{\mathsf{dom}\,\Gamma} = \bar{\alpha}'\big|_{\mathsf{dom}\,\Gamma} \implies \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S)) = \mathcal{C}_{(\bar{\alpha}',\alpha,\dot{\alpha})}((\Gamma, e, S)) \qquad \text{(C-cong)}$$

$$\bar{\alpha}\ x = \bot \implies \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S)) = \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma \setminus \{x\}, e, S)) \qquad \text{(C-not-called)}$$

Furthermore, the cardinality analysis is likewise a forward analysis and has to be conservative about function arguments:

$$\$x \in S \implies [x \mapsto \infty] \sqsubseteq \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S)) \qquad \text{(C-args)}$$

The prognosis may ignore update markers on the stack:

$$\mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, \#x{\cdot}S)) \sqsubseteq \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S)) \qquad \text{(C-upd)}$$

An imminent call is prognosed:

$$[x \mapsto \mathbf{1}] \sqsubseteq \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, x, S)) \qquad \text{(C-call)}$$

Evaluation improves the prognosis: Note that in (C-Var$_1$) and (C-Var$'_1$), we account for the call to x with the $-$ operator.

$$\mathcal{C}_{(\bar{\alpha},\alpha+1,\dot{\alpha})}((\Gamma, e, \$x{\cdot}S)) \sqsubseteq \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e\ x, S)) \qquad \text{(C-App)}$$

$$\mathcal{C}_{(\bar{\alpha},\alpha-1,\dot{\alpha})}((\Gamma, e[y := x], S)) \sqsubseteq \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, \lambda y.\ e, \$x{\cdot}S)) \qquad \text{(C-Lam)}$$

$$(x \mapsto e) \in \Gamma,\ \neg\,\mathsf{isVal}(e) \implies \mathcal{C}_{(\bar{\alpha},\bar{\alpha}\ x,\dot{\alpha})}((\Gamma \setminus \{x\}, e, \#x{\cdot}S)) \sqsubseteq \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, x, S)) - x \qquad \text{(C-Var}_1\text{)}$$

$$(x \mapsto e) \in \Gamma,\ \mathsf{isVal}(e) \implies \mathcal{C}_{(\bar{\alpha},\bar{\alpha}\ x,\dot{\alpha})}((\Gamma, e, S)) \sqsubseteq \mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, x, S)) - x \qquad \text{(C-Var}'_1\text{)}$$

$$\mathsf{isVal}(e) \implies \mathcal{C}_{(\bar{\alpha},0,\dot{\alpha})}((\Gamma[x \mapsto e], e, S)) \sqsubseteq \mathcal{C}_{(\bar{\alpha},0,\dot{\alpha})}((\Gamma, e, \#x{\cdot}S)) \qquad \text{(C-Var}_2\text{)}$$

$$\mathcal{C}_{(\bar\alpha,0,\alpha\cdot\dot\alpha)}((\Gamma, e, (e_{\mathbf{t}} : e_{\mathbf{f}})\cdot S)) \sqsubseteq \mathcal{C}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, e\,?\,e_{\mathbf{t}} : e_{\mathbf{f}}, S)) \tag{C-If$_1$}$$

$$b \in \{\mathbf{t}, \mathbf{f}\} \implies \mathcal{C}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, e_b, S)) \sqsubseteq \mathcal{C}_{(\bar\alpha,0,\alpha\cdot\dot\alpha)}((\Gamma, \mathbf{C}_b, (e_{\mathbf{t}} : e_{\mathbf{f}})\cdot S)) \tag{C-If$_2$}$$

The specification for the **let**-bindings connects the arity analysis, the cardinality analysis and the cardinality prognosis:

$$\mathsf{dom}\,\Delta \cap \mathsf{fv}(\Gamma, S) = \{\},\ \mathsf{dom}\,\bar\alpha \subseteq \mathsf{dom}\,\Gamma \cup \#S \implies \mathcal{C}_{(\mathcal{A}_\alpha(\Delta,e)\sqcup\bar\alpha,\alpha,\dot\alpha)}((\Delta\cdot\Gamma, e, S)) \sqsubseteq \mathcal{C}_\alpha(\Delta, e) \sqcup \mathcal{C}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, \mathbf{let}\ \Delta\ \mathbf{in}\ e, S)) \tag{C-Let}$$

Finally, we need an equivalent to Definition 3 that now restricts the arity analysis only for thunks that might be called more than once:

$$x \in \mathsf{thunks}\,\Gamma,\ \mathcal{C}_\alpha(\Gamma, e)\ x = \infty \implies \mathcal{A}_\alpha(\Gamma, e)\ x = 0 \tag{Ah-∞-thunk}$$

□

Safety The safety proof proceeds similarly to the one for Lemma 2. But now we are allowed to eta-expand thunks that are called at most once. This has considerable technical implications for the proof:

- An eta-expanded expression is a value, so in the transformed program, VAR$_2$ occurs immediately after VAR$_1$. In the original program, however, an update marker stays on the stack until the expression is evaluated to a value, and then VAR$_2$ fires without a correspondence in the evaluation of the transformed program. In particular, the update marker can interfere with uses of Lemma 1.
- Because the eta-expanded expression is a value, it stays on the heap as it is, whereas in the original program, it is first evaluated. Evaluation can reduce the number of free variables of the expression, so subsequent choices of fresh variables in LET$_1$ in the original evaluation might not be suitable in the evaluation of the transformed program.

A more complicated variant of Lemma 1 and carrying a variable renaming around throughout the proof might solve these problems, but would complicate it too much. We therefore apply a small trick and simply allow unwanted update markers to disappear, by defining a variant of the semantics:

Definition 7 (Forgetful semantics) The relation $\Rightarrow_\#$ is defined by

$$(\Gamma, e, S) \Rightarrow (\Gamma', e', S') \implies (\Gamma, e, S) \Rightarrow_\# (\Gamma', e', S').$$

and

$$(\Gamma, e, \#x\cdot S) \Rightarrow_\# (\Gamma, e, S) \qquad \text{DROPUPD}$$

□

This way, a one-shot binding can disappear completely after it has been called, making it easier to relate the original program to the transformed program. Because $\Rightarrow_\#$ contains $\Rightarrow$, Lemma 1 holds here as well. Afterwards, and outside the scope of the safety proof, we will recover the original semantics from the forgetful semantics.

In the proof we keep track of the set of removed bindings (named r), and write $(\Gamma, e, S) - r := (\Gamma \setminus r, e, S - r)$ for the configuration with bindings from the set r removed. The stack $(S - r)$ is S without update markers $\#x$ where $x \in r$.

We also keep track of $\bar\gamma$: $\mathsf{Var} \to \mathsf{Card}$, the current cardinalities of the variables on the heap:

Definition 8 (Cardinality arity annotation consistency) We write $(\bar\alpha, \alpha, \dot\alpha, \bar\gamma, r) \rhd_{\mathbf{C}} (\Gamma, e, S)$, iff

- the arity information is consistent, $(\bar\alpha, \alpha, \dot\alpha) \rhd (\Gamma, e, S) - r$,
- $\mathsf{dom}\,\bar\alpha = \mathsf{dom}\,\bar\gamma$,
- the cardinality information is correct, $\mathcal{C}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, e, S)) \sqsubseteq \bar\gamma$,
- many-called thunks are not going to be eta-expanded, i.e. $\bar\alpha\ x = 0$ for $x \in \mathsf{thunks}\,\Gamma$ with $\bar\gamma\ x = \infty$ and
- only bindings that are not going to be called ($\bar\gamma\ x = \bot$) are removed, i.e. $r \subseteq (\mathsf{dom}\,\Gamma \cup \#S) - \mathsf{dom}\,\bar\gamma$. □

Lemma 3 *Assume $\mathcal{A}$ and $\mathcal{C}$ fulfill the specifications in Definitions 2 and 6.*

If $(\Gamma, e, S) \Rightarrow^* (\Gamma', e', S')$ and $(\bar\alpha, \alpha, \dot\alpha, \bar\gamma, r) \rhd_{\mathbf{C}} (\Gamma, e, S)$, then there exists $(\bar\alpha', \alpha', \dot\alpha', \bar\gamma', r')$ such that $(\bar\alpha', \alpha', \dot\alpha', \bar\gamma', r') \rhd_{\mathbf{C}} (\Gamma', e', S')$, and $\mathsf{T}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, e, S) - r) \Rightarrow_\#^* \mathsf{T}_{(\bar\alpha',\alpha',\dot\alpha')}((\Gamma', e', S') - r')$. □

The lemma is an analog to Lemma 2. The main difference, besides the extra data to keep track of, is that we produce an evaluation in the forgetful semantics, with some bindings removed.

PROOF by the individual steps of $\Rightarrow^*$. The preservation of the arity annotation consistency in the proof of Lemma 2 can be used here as well. Note that both the arity annotation requirement and the transformation are applied to $(\Gamma, e, S) - r$, so this goes well together. The correctness of the cardinality information (the second condition in Definition 8) follows easily from the inequalities in Definition 6.

We elaborate only on the interesting cases:

Case VAR$_1$: We cannot have $\bar\gamma\ x = \bot$ because of (C-call).

If $\bar\gamma\ x = \infty$ we get $\bar\alpha\ x = 0$, as before, and nothing surprising happens.

If $\bar\gamma\ x = \mathbf{1}$, we know that this is the only call to x, so we set $r' = r \cup \{x\}$, $\bar\gamma' = \bar\gamma - x$ and use DROPUPD to get rid of the mention of $\#x$ on the stack.

Case VAR$_2$: If $x \notin r$, proceed as before. If $x \in r$, then the transformed configurations are identical and the $\Rightarrow_\#^*$ judgment follows from reflexivity. ■

Corollary 2 *The cardinality based arity analysis is safe for closed expressions, i.e. if $\mathsf{fv}\,e = \{\}$ and $([], e, []) \Rightarrow^* (\Gamma, v, [])$ then there exists Γ' and v' such that $([], \mathsf{T}_0(e), []) \Rightarrow^* (\Gamma', v', [])$ where Γ and Γ' contain the same number of bindings.* □

PROOF We need $\mathsf{fv}\,e = \{\}$ to have $\mathcal{C}_{\bot,0,[]}(([], e, [])) = \bot$, so that $(\bot, 0, [], \bot, []) \rhd_{\mathbf{C}} ([], e, [])$ holds. Now Lemma 2 gives us $\bar\alpha$, α, $\dot\alpha$ and r so that $\mathsf{T}_{(\bot,0,[])}(([], e, [])) \Rightarrow_\#^* \mathsf{T}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, v, []) - r)$.

As the forgetful semantics only drops unused bindings, but does not otherwise behave any different than the real semantics, a technical lemma allows us to recover $\mathsf{T}_{(\bot,0,[])}(([], e, [])) \Rightarrow^* \mathsf{T}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma', v, []))$ for a Γ' where $\overline{\mathsf{T}}_{\bar\alpha}(\Gamma) - r = \Gamma' - r'$. As $r \subseteq \Gamma$ and $r' \subseteq \Gamma'$, this concludes the proof of the corollary: Γ, $\overline{\mathsf{T}}_{\bar\alpha}(\Gamma)$ and Γ' all bind the same variables. ■

5.2 Trace tree cardinality analysis

In the second refinement, we look – still quite abstractly – at the implementation of the cardinality analysis. For the arity information, the type of the result required for the transformation ($\mathsf{Var} \to \mathbb{N}_\bot$) was sufficiently rich to be used in the analysis as well. This is unfortunately not the case for the cardinality analysis: Even if we know that an expression calls x and y each at most once, this does not tell us whether these calls can occur together (as in $e\ x\ y$) or whether they are exclusive (as in $e\,\mathbf{?}\,x : y$).

So we need a richer type that captures the future calls of an expression, can distinguish different code paths and maps easily to $\mathsf{Var} \to \mathsf{Card}$: The type TTree of (possibly infinite) trees, where each edge is labeled with variable name, and a node has at most one outgoing edge for each variable name. The paths in the tree correspond to the possible executions and the labels on the edges record each occurring variable call. We use t for values of type TTree.

There are other, equivalent ways to interpret this type: Each TTree corresponds to a non-empty set of (finite) lists of variable names that is prefixed-closed (i.e. for every list in the set, its prefixes are also in the set). Each such list corresponds to a (finite) path in the tree. The function $\mathsf{paths} \colon \text{TTree} \to 2^{[\mathsf{Var}]}$ implements this correspondence.

Another view is given by the function

$$\mathsf{next} \colon \mathsf{Var} \to \text{TTree} \to \text{TTree}_\bot,$$

where $\mathsf{next}\ \ x\ t = t'$ iff the root of t has an edge labeled x leading to t', and $\mathsf{next}\ \ x\ t = \bot$ if the root of t has no edge labeled x. In that sense, TTree represents automata with labeled transitions.

The basic operations on trees are $\oplus$, given by $\mathsf{paths}(t \oplus t') = \mathsf{paths}\,t \cup \mathsf{paths}\,t'$, and $\otimes$, where $\mathsf{paths}(t \otimes t')$ is the set of all interleavings of lists from $\mathsf{paths}\,t$ with lists from $\mathsf{paths}\,t'$. We write t^* for $t \otimes t \otimes t \otimes \cdots$. A tree is called *repeatable* if $t = t \otimes t = t^*$.

The partial order used on TTree is $t \sqsubseteq t' \iff \mathsf{paths}\ t \subseteq \mathsf{paths}\ t'$. We write $\bullet$ for the tree with no edges and x for $\bullet\!\xrightarrow{x}$, the tree with exactly one edge labeled x. The tree $t \setminus V$ is t with all edges with labels in V contracted, $t|_V$ is t with all edges but those labeled with variables in V contracted.

If we have a binding (Γ, e), and for e as well as for all bound expressions, we have a TTree describing their calls, how would we combine that information? A first attempt might be a function $s \colon (\mathsf{Var} \to \text{TTree}) \to \text{TTree} \to \text{TTree}$ defined by

$$\mathsf{next}\ \ x\ (s\ \bar{t}\ t) := \begin{cases} \bot & \text{if } \mathsf{next}\ \ x\ t = \bot \\ s\ \bar{t}\ (t' \otimes \bar{t}\ x) & \text{if } \mathsf{next}\ \ x\ t = t', \end{cases}$$

that traverses the tree t and upon every call interleaves the tree of the called name, $\bar{t}\ x$, with the remainder of t.

This is a good start, but it does not cater for thunks, where the first call behaves differently than later calls. Therefore, we have to tell s which variables are bound to thunks, and give them special treatment: After a variable x referring to a thunk is evaluated, we pass on a modified map where $\bar{t}\ x = \bullet$.

Hence $s \colon 2^{\mathsf{Var}} \to (\mathsf{Var} \to \text{TTree}) \to \text{TTree} \to \text{TTree}$ is defined by

$$\begin{aligned} &\mathsf{next}\ \ x\ (s_T\ \bar{t}\ t) \\ &:= \begin{cases} \bot & \text{if } \mathsf{next}\ \ x\ t = \bot \\ s_T\ \bar{t}\ (t' \otimes \bar{t}\ x) & \text{if } \mathsf{next}\ \ x\ t = t', x \notin T \\ s_T\ (\bar{t}[x \mapsto \bullet])\ (t' \otimes \bar{t}\ x) & \text{if } \mathsf{next}\ \ x\ t = t', x \in T. \end{cases} \end{aligned}$$

The ability to define this function (relatively) easily is the main advantage of working with trace trees instead of co-call graphs at this stage.

We project a TTree to a value of type $(\mathsf{Var} \to \mathsf{Card})$, as required for a cardinality analysis, using $c \colon \text{TTree} \to (\mathsf{Var} \to \mathsf{Card})$ defined by

$$c(t)\ x := \begin{cases} \bot, & \text{if } x \text{ does not occur in } t \\ \mathbf{1}, & \text{if on each path in } t,\ x \text{ occurs at most once} \\ \infty, & \text{otherwise.} \end{cases}$$

Specification A tree cardinality analysis determines for every expression e and arity α the tree $\mathcal{T}_\alpha(e)$ of calls to free variables of e which are performed by evaluating e with α arguments and using the result in any way.

We write $\overline{\overline{\mathcal{T}}}_{\bar{\alpha}}(\Gamma)$ for the analysis lifted to bindings, returning $\bot$ for variables not bound in Γ or mapped to $\bot$ in $\bar{\alpha}$.

We also need a variant $\mathcal{T}_\alpha(\Gamma, e)$ that, given bindings Γ, an expression e and an arity α, reports the calls on $\mathsf{dom}\,\Gamma$ performed by e and Γ with these bindings in scope.

We can now identify conditions on $\mathcal{T}$ that allow us to satisfy the specifications in Definition 6.

Definition 9 (Tree cardinality analysis specification) We expect the cardinality analysis to agree with the arity analysis on which variables are called at all:

$$\mathsf{dom}\,\mathcal{T}_\alpha(e) = \mathsf{dom}\,\mathcal{A}_\alpha(e) \qquad \text{(T-dom)}$$

$$\mathsf{dom}\,\mathcal{T}_\alpha(\Gamma, e) = \mathsf{dom}\,\mathcal{A}_\alpha(\Gamma, e) \qquad \text{(Th-dom)}$$

Inequalities for the syntactic constructs:

$$x^* \otimes \mathcal{T}_{\alpha+1}(e) \sqsubseteq \mathcal{T}_\alpha(e\ x) \qquad \text{(T-App)}$$

$$(\mathcal{T}_{\alpha-1}(e)) \setminus \{x\} \sqsubseteq \mathcal{T}_\alpha(\lambda x.\, e) \qquad \text{(T-Lam)}$$

$$\mathcal{T}_\alpha(e[y := x]) \sqsubseteq x^* \otimes (\mathcal{T}_\alpha(e)) \setminus \{y\} \qquad \text{(T-subst)}$$

$$x \sqsubseteq \mathcal{T}_\alpha(x) \qquad \text{(T-Var)}$$

$$\mathcal{T}_0(e) \otimes (\mathcal{T}_\alpha(e_{\mathbf{t}}) \oplus \mathcal{T}_\alpha(e_{\mathbf{f}})) \sqsubseteq \mathcal{T}_\alpha(e\,\mathbf{?}\,e_{\mathbf{t}} : e_{\mathbf{f}}) \qquad \text{(T-If)}$$

$$(s_{\mathsf{thunks}\,\Gamma}\ (\overline{\overline{\mathcal{T}}}_{\mathcal{A}_\alpha(\Gamma,e)}(\Gamma))\ (\mathcal{T}_\alpha(e))) \setminus \mathsf{dom}\,\Gamma \sqsubseteq \mathcal{T}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e)$$
$$\text{(T-Let)}$$

For values, analyzed without arguments, the analysis is expected to return a repeatable tree:

$$\mathsf{isVal}(e) \implies \mathcal{T}_0(e) \text{ is repeatable} \qquad \text{(T-value)}$$

The specification for $\mathcal{A}_\alpha(\Gamma, e)$ is closely related to (T-Let):

$$(s_{\mathsf{thunks}\,\Gamma}\ (\overline{\overline{\mathcal{T}}}_{\mathcal{A}_\alpha(\Gamma,e)}(\Gamma))\ (\mathcal{T}_\alpha(e)))|_{\mathsf{dom}\,\Gamma} \sqsubseteq \mathcal{T}_\alpha(\Gamma, e) \qquad \text{(Th-s)}$$

And finally, the connection to the arity analysis:

$$x \in \mathsf{thunks}\,\Gamma,\ c(\mathcal{T}_\alpha(\Gamma, e))\ x = \infty \implies (\mathcal{A}_\alpha(\Gamma, e))\ x = 0$$
$$\text{(Th-}\infty\text{-thunk)}$$

□

Safety If we have a tree cardinality analysis, we can define a cardinality analysis in the sense of the previous section. The definition for $\mathcal{C}_\alpha(\Gamma, e)$ is straight forward:

$$\mathcal{C}_\alpha(\Gamma, e) := c(\mathcal{T}_\alpha(\Gamma, e)).$$

In order to define $\mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S))$ we need to fold the tree cardinality analysis over the stack:

$$\begin{aligned} \dot{\mathcal{T}}_\_([]) &:= \bot \\ \dot{\mathcal{T}}_{\alpha\cdot\dot{\alpha}}((e_{\mathbf{t}} : e_{\mathbf{f}})\cdot S) &:= \dot{\mathcal{T}}_{\dot{\alpha}}(S) \otimes (\mathcal{T}_\alpha(e_{\mathbf{t}}) \oplus \mathcal{T}_\alpha(e_{\mathbf{f}})) \\ \dot{\mathcal{T}}_{\dot{\alpha}}(\$x\cdot S) &:= \dot{\mathcal{T}}_{\dot{\alpha}}(S) \otimes x^* \\ \dot{\mathcal{T}}_{\dot{\alpha}}(\#x\cdot S) &:= \dot{\mathcal{T}}_{\dot{\alpha}}(S). \end{aligned}$$

With this we can define

$$\mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S)) := c\big(s_{\mathsf{thunks}\,\Gamma}\ (\overline{\overline{\mathcal{T}}}_{\bar{\alpha}}(\Gamma))\ (\mathcal{T}_\alpha(e) \otimes \dot{\mathcal{T}}_{\dot{\alpha}}(S))\big),$$

and set out to prove

Lemma 4 *Given a tree cardinality analysis satisfying Definition 9, together with an arity analysis satisfying Definition 2, the derived cardinality analysis satisfies Definition 6.* □

PROOF The proof follows by calculation involving c and the operations on trees.

Condition (C-If$_2$) is where the precision comes from, as we retain the knowledge that two code paths are mutually exclusive.

In the proof for (C-Var$_2$), we know that $\mathcal{T}_0(e)$ is repeatable, as $\mathsf{isVal}(e)$. This allows us to use that if a repeatable tree t is already contained in the second argument to s, then we can remove it from the range of the first argument:

$$s_T\ (\bar{t}[x \mapsto t])\ (t \otimes t') = s_T\ \bar{t}\ (t \otimes t') \qquad \blacksquare$$

5.3 Co-Call cardinality analysis

The preceding section provides a framework for a cardinality analysis, but the infinite nature of the TTree data type prevents an implementation on that level. For a real implementation, we need a practically implementable data type that approximates the trees.

The data type Graph used in the implementation is an undirected, non-transitive graph with loops on the set of variables. The intuition is that only the nodes of G (denoted by $\mathsf{dom}\,G$) are called, and that an edge x—$y \in G$ indicates that x and y can be called together, while the absence of an edge guarantees that calls to x resp. y are mutually exclusive.

Loops thus indicate whether a variable is going to be called more than once: The graph x——y allows at most one call to y (possibly together with one call to x), while x——$y\supset$ allows any number of calls to y (but still at most one to x).

We specify graphs via their edge sets, e.g.

$$V \times V' := \{x\text{—}y \mid x \in V \wedge y \in V' \vee y \in V \wedge x \in V'\}$$

for the Cartesian product of variable sets, and either specify their node set separately (e.g. $\mathsf{dom}(V \times V') = \mathsf{dom}\,V \cup \mathsf{dom}\,V'$) or leave it implicit.

We write $V^2 := V \times V$. The set of neighbors of a variable is $N_x(G) := \{y \mid x\text{—}y \in G\}$. The graph $G \setminus V$ is G with nodes in V removed, while $G\big|_V$ is G with only nodes in V retained. The graphs are ordered by inclusion, with $\bot = \{\}$.

We can convert a graph to a TTree with $t\colon$ Graph $\to$ TTree:

$$\begin{aligned}&\mathsf{paths}(t(G))\\ &\quad := \{x_1 \cdots x_n \mid \forall i.\, x_i \in \mathsf{dom}\,G \wedge \forall j \neq i.\, x_i\text{—}x_j \in G\}.\end{aligned}$$

We can also approximate a TTree by a Graph with the function $g\colon$ TTree $\to$ Graph:

$$g(t) := \bigsqcup\{\dot{g}(\dot{x}) \mid \dot{x} \in \mathsf{paths}\,t\}$$

using $\dot{g}\colon$ [Var] $\to$ Graph where $\mathsf{dom}\,\dot{g}(x_1 \cdots x_n) = \{x_1, \ldots, x_n\}$ and $\dot{g}(x_1 \cdots x_n) := \{x_i\text{—}x_j \mid i \neq j \leq n\}$.

The mappings t and g form a monotone Galois connection: $g(t) \sqsubseteq G \iff t \sqsubseteq t(G)$. It even is a Galois insertion, as $g(t(G)) = G$.

Specification We proceed in the usual scheme, by giving a specification for a safe co-call cardinality analysis, connecting it to the tree cardinality analysis, and eventually proving that our implementation fulfills the specification.

A co-call cardinality analysis determines for each expression e and incoming arity α its co-call graph $\mathcal{G}_\alpha(e)$. As before, we also require a variant that analyses bindings, written $\mathcal{G}_\alpha(\Gamma, e)$. The conditions in the following definition are obviously designed to connect to Definition 9.

Definition 10 (Co-call cardinality analysis specification) We want the co-call graph analysis to agree with the arity analysis on what is called at all:

$$\mathsf{dom}\,\mathcal{G}_\alpha(e) = \mathsf{dom}\,\mathcal{A}_\alpha(e) \qquad \text{(G-dom)}$$

As usual, we have inequalities for the syntactic constructs:

$$\mathcal{G}_{\alpha+1}(e) \cup (\{x\} \times \mathsf{fv}(e\ x)) \sqsubseteq \mathcal{G}_\alpha(e\ x) \qquad \text{(G-App)}$$

$$\mathcal{G}_{\alpha-1}(e) \setminus \{x\} \sqsubseteq \mathcal{G}_\alpha(\lambda x.\, e) \qquad \text{(G-Lam)}$$

$$\mathcal{G}_\alpha(e[y := x]) \setminus \{x, y\} \sqsubseteq \mathcal{G}_\alpha(e) \setminus \{x, y\} \quad \text{(G-subst)}$$

$$\begin{aligned}&\mathcal{G}_0(e) \cup \mathcal{G}_\alpha(e_{\mathbf{t}}) \cup \mathcal{G}_\alpha(e_{\mathbf{f}}) \cup\\ &(\mathsf{dom}\,\mathcal{A}_0(e) \times (\mathsf{dom}\,\mathcal{A}_\alpha(e_{\mathbf{t}}) \cup \mathsf{dom}\,\mathcal{A}_\alpha(e_{\mathbf{f}})))\\ &\qquad\qquad \sqsubseteq \mathcal{G}_\alpha(e\ \mathbf{?}\ e_{\mathbf{t}} : e_{\mathbf{f}}) \qquad \text{(G-If)}\end{aligned}$$

$$\mathcal{G}_\alpha(\Gamma, e) \setminus \mathsf{dom}\,\Gamma \sqsubseteq \mathcal{G}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e) \qquad \text{(G-Let)}$$

$$\mathsf{isVal}(e) \implies \qquad (\mathsf{fv}\,e)^2 \sqsubseteq \mathcal{G}_0(e) \qquad \text{(G-value)}$$

The following conditions concern $\mathcal{G}_\alpha(\Gamma, e)$, which has to cater for the calls originating in e,

$$\mathcal{G}_\alpha(e) \sqsubseteq \mathcal{G}_\alpha(\Gamma, e), \qquad \text{(Gh-body)}$$

the calls originating in the right-hand-sides,

$$(x \mapsto e') \in \Gamma \implies \quad \mathcal{G}_{\mathcal{A}_\alpha(\Gamma,e)\ x}(e') \sqsubseteq \mathcal{G}_\alpha(\Gamma, e), \qquad \text{(Gh-heap)}$$

and finally the extra edges between what is called from the right-hand-side of a variable and whatever the variable is called with:

$$\begin{aligned}&(x \mapsto e') \in \Gamma,\ \mathsf{isVal}(e') \implies\\ &\qquad (\mathsf{fv}\,e') \times N_x(\mathcal{G}_a(\gamma, e)) \sqsubseteq \mathcal{G}_\alpha(\Gamma, e). \qquad \text{(Gh-extra)}\end{aligned}$$

For thunks, we can be slightly more precise: Only one call to them matters, so we can ignore a possible edge x—x:

$$\begin{aligned}&(x \mapsto e') \in \Gamma,\ \neg\,\mathsf{isVal}(e') \implies\\ &\quad (\mathsf{fv}\,e') \times (N_x(\mathcal{G}_a(\gamma, e)) \setminus \{x\}) \sqsubseteq \mathcal{G}_\alpha(\Gamma, e) \qquad \text{(Gh-extra')}\end{aligned}$$

Finally, we need to ensure that the cardinality analysis is actually used by the arity analysis when dealing with thunks. For recursive bindings, we never eta-expand thunks:

$$\begin{aligned}&\mathsf{rec}\,\Gamma,\ x \in \mathsf{thunks}\,\Gamma,\ x \in \mathsf{dom}\,\mathcal{A}_\alpha(\Gamma, e) \implies\\ &\qquad\qquad \mathcal{A}_\alpha(\Gamma, e) = 0 \qquad \text{(Rec-}\infty\text{-thunk)}\end{aligned}$$

But for a non-recursive thunk, we only have to worry about thunks which are possibly called multiple times:

$$\begin{aligned}&x \notin \mathsf{fv}\,e',\ \neg\,\mathsf{isVal}(e'),\ x\text{—}x \in \mathcal{G}_\alpha(\Gamma, e) \implies\\ &\qquad\qquad \mathcal{A}_\alpha([x \mapsto e'], e) = 0 \qquad \text{(Nonrec-}\infty\text{-thunk)}\end{aligned}$$

□

Safety From a co-call analysis fulfilling Definition 10 we can derive a tree cardinality analysis fulfilling Definition 9, using

$$\mathcal{T}_\alpha(e) := t(\mathcal{G}_\alpha(e)).$$

The definition of $\mathcal{T}_\alpha(\Gamma, e)$ differs for nonrecursive and recursive bindings. For a non-recursive binding $\Gamma = [x \mapsto e']$ we have $\mathcal{T}_\alpha(\Gamma, e) := t(\mathcal{G}_\alpha(e))\big|_{\mathsf{dom}\,\Gamma}$ and for recursive Γ we define $\mathcal{T}_\alpha(\Gamma, e) := t((\mathsf{dom}\,\mathcal{A}_\alpha(\Gamma, e))^2)$, i.e. the bound variables may call each other in any way.

Lemma 5 *Given a co-call cardinality analysis satisfying Definition 10, together with an arity analysis satisfying Definition 2, the derived cardinality analysis satisfies Definition 9.* □

PROOF Most conditions of Definition 9 follow by simple calculation from their counterpart in Definition 10 using the

Galois connection

$$t \sqsubseteq t(G) \iff g(t) \sqsubseteq G$$

and identities such as $g(t \oplus t') = g(t) \cup g(t')$ and $g(t \otimes t') = g(t) \cup g(t') \cup (\mathsf{dom}\, t \times \mathsf{dom}\, t')$.

For (T-Let), we use (G-Let) with the following lemma:

$$\begin{gathered} g(t) \sqsubseteq G \qquad \forall x \notin S.\, \bar{t}\, x = \bot \qquad \forall x \in S.\, g(\bar{t}\, x) \sqsubseteq G \\ \forall x \in S,\, x \notin T.\ \mathsf{dom}(\bar{t}\, x) \times N_x(G) \sqsubseteq G \\ \forall x \in S,\, x \in T.\ \mathsf{dom}(\bar{t}\, x) \times (N_x(G) \setminus \{x\}) \sqsubseteq G \\ \implies g((s_T\, \bar{t}\, t) \setminus S) \sqsubseteq G, \end{gathered}$$

which we instantiate with $T = \mathsf{thunks}\,\Gamma$, $\bar{t} = \overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Gamma,e)}(\Gamma)$, $t = \mathcal{T}_\alpha(e)$ and $S = \mathsf{dom}\,\Gamma$. ∎

5.4 Call Arity, concretely

At last we can give the complete and concrete co-call analysis corresponding to GHC's Call Arity, and establish its safety via our chain of refinements, simply by checking the conditions in Definition 10.

The arity analysis is:

$$\begin{aligned} \mathcal{A}_\alpha(x) &:= [x \mapsto \alpha] \\ \mathcal{A}_\alpha(e\, x) &:= \mathcal{A}_{\alpha+1}(e) \sqcup [x \mapsto 0] \\ \mathcal{A}_\alpha(\lambda x.\, e) &:= \mathcal{A}_{\alpha-1}(e) \setminus \{x\} \\ \mathcal{A}_\alpha(e \,?\, e_{\mathbf{t}} : e_{\mathbf{f}}) &:= \mathcal{A}_0(e) \sqcup \mathcal{A}_\alpha(e_{\mathbf{t}}) \sqcup \mathcal{A}_\alpha(e_{\mathbf{f}}) \\ \mathcal{A}_\alpha(\mathbf{C}_b) &:= \bot \qquad \text{for } b \in \{\mathbf{t}, \mathbf{f}\} \end{aligned}$$

The analysis of a let expression $\mathcal{A}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e)$ as well as the analysis of a binding $\mathcal{A}_\alpha(\Gamma, e)$ are defined differently for recursive and non-recursive bindings.

For a recursive Γ, we have $\mathcal{A}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e) := \bar{\alpha} \setminus \mathsf{dom}\,\Gamma$ and $\mathcal{A}_\alpha(\Gamma, e) := \bar{\alpha}|_{\mathsf{dom}\,\Gamma}$ where $\bar{\alpha}$ is the least fixed point defined by the equation

$$\bar{\alpha} = \overline{\mathcal{A}}_{\bar{\alpha}}(\Gamma) \sqcup \mathcal{A}_\alpha(e) \sqcup [x \mapsto 0 \mid x \in \mathsf{thunks}\,\Gamma].$$

For a non-recursive binding $\Gamma = [x \mapsto e']$ we have $\mathcal{A}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e) := (\mathcal{A}_{\alpha'}(e') \sqcup \mathcal{A}_\alpha(e)) \setminus \mathsf{dom}\,\Gamma$ and $\mathcal{A}_\alpha(\Gamma, e) := [x \mapsto \alpha']$ where

$$\alpha' := \begin{cases} 0 & \text{if } \neg\, \mathsf{isVal}(e') \text{ and } x\text{—}x \in \mathcal{G}_\alpha(e) \\ \mathcal{A}_\alpha(e)\ x & \text{otherwise.} \end{cases}$$

We have $\mathsf{dom}\,\mathcal{G}_\alpha(e) = \mathsf{dom}\,\mathcal{A}_\alpha(e)$ and

$$\begin{aligned} \mathcal{G}_\alpha(x) &:= \{\} \\ \mathcal{G}_\alpha(e\, x) &:= \mathcal{G}_{\alpha+1}(e) \cup (\{x\} \times \mathsf{fv}(e\, x)) \\ \mathcal{G}_0(\lambda x.\, e) &:= (\mathsf{fv}\, e)^2 \setminus \{x\} \\ \mathcal{G}_{\alpha+1}(\lambda x.\, e) &:= \mathcal{G}_\alpha(e) \setminus \{x\} \\ \mathcal{G}_\alpha(e \,?\, e_{\mathbf{t}} : e_{\mathbf{f}}) &:= \mathcal{G}_0(e) \cup \mathcal{G}_\alpha(e_{\mathbf{t}}) \cup \mathcal{G}_\alpha(e_{\mathbf{f}}) \cup \\ &\quad\ (\mathsf{dom}\,\mathcal{A}_0(e) \times (\mathsf{dom}\,\mathcal{A}_\alpha(e_{\mathbf{t}}) \cup \mathsf{dom}\,\mathcal{A}_\alpha(e_{\mathbf{f}}))) \\ \mathcal{G}_\alpha(\mathbf{C}_b) &:= \{\} \qquad \text{for } b \in \{\mathbf{t}, \mathbf{f}\} \\ \mathcal{G}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e) &:= \mathcal{G}_\alpha(\Gamma, e) \setminus \mathsf{dom}\,\Gamma \end{aligned}$$

The analysis result for bindings is different for recursive and non-recursive bindings and uses the auxiliary function

$$\mathcal{G}_{\bar{\alpha};G}(x \mapsto e') := \begin{cases} (\mathsf{fv}\, e')^2 & \text{if } \mathsf{isVal}(e') \wedge x\text{—}x \in G \\ \mathcal{G}_{\bar{\alpha}\ x}(e') & \text{otherwise,} \end{cases}$$

which calculates the co-calls of an individual binding, adding the extra edges between multiple invocations of a bound variable, unless it is bound to a thunk and hence shared.

For recursive Γ we define $\mathcal{G}_\alpha(\Gamma, e)$ as the least fixed point fulfilling

$$\begin{aligned} \mathcal{G}_\alpha(\Gamma, e) = \mathcal{G}_\alpha(e) &\sqcup \bigsqcup_{(x \mapsto e') \in \Gamma} \mathcal{G}_{\mathcal{A}_\alpha(\Gamma,e);\mathcal{G}_\alpha(\Gamma,e)}(x \mapsto e') \\ &\sqcup \bigsqcup_{(x \mapsto e') \in \Gamma} (\mathsf{fv}\, e' \times N_x(\mathcal{G}_\alpha(\Gamma, e))). \end{aligned}$$

For a non-recursive $\Gamma = [x \mapsto e']$, we have

$$\begin{aligned} \mathcal{G}_\alpha(\Gamma, e) = \mathcal{G}_\alpha(e) &\sqcup \mathcal{G}_{\mathcal{A}_\alpha(\Gamma,e);\mathcal{G}_\alpha(e)}(x \mapsto e') \\ &\sqcup \begin{cases} \mathsf{fv}\, e' \times (N_x(\mathcal{G}_\alpha(e)) \setminus \{x\}) & \text{if } \neg\, \mathsf{isVal}(e') \\ \mathsf{fv}\, e' \times N_x(\mathcal{G}_\alpha(e)) & \text{if } \mathsf{isVal}(e'). \end{cases} \end{aligned}$$

Theorem 1 *Call Arity is safe (in the sense of Definition 1).*

PROOF By straightforward calculation (and simple induction for (G-subst)), we can show that the analyses fulfill Definition 2 and Definition 10. So by Lemma 5, Lemma 4, Lemma 3 and Corollary 1, the analyses are safe. ∎

6. The formalization in Isabelle

On their own, the proofs presented in the previous sections are not very interesting, as they are neither very elegant nor very innovative. What sets them apart from similar work is that these proofs have been carried out in the interactive theorem prover Isabelle [22]. This provides a level of assurance that is hard to reach using pen-and-paper-proofs.

But it also greatly increases the effort involved in obtaining a result like Theorem 1. The Isabelle development corresponding to this paper, including the definition of the syntax and the semantics, contains roughly 12,000 lines of code with 1,200 lemmas (many small, some large) in 75 theories, created over the course of 9 months [4]. Large parts of it, however, can be re-used for other developments: The syntax and semantics, of course, but also the newly created data types like the trace trees and the co-call graphs.

Much of the complexity is owed to the problem of bindings. Using Nominal logic ([27], implemented for Isabelle in Christian Urban's Nominal2 package) helped a lot here, but still incurs a technical overhead, as all involved definitions have to be proven equivariant, i.e. oblivious to variable renaming. While usually simple to prove, these lemmas still have to be stated.

Another cause of overhead is ensuring that all analyses and the operators used by them are monotone and continuous, so that the fixed points are actually well-defined. Here, the HOLCF package by Brian Huffman [14] is used with good results, but again not without an extra cost compared to handwaving over such issues in pen-and-paper proofs.

So while the actual result shown here might not have warranted that effort on its own – after all, performance regressions due to bugs in the Call Arity analysis do not have very serious consequences – it lays ground towards formalizing more and more parts of the core data structures and algorithms in our compilers.

The separation into individual theories (Isabelle's equivalent to Haskell's modules) as well as the use of *locales* ([2], Isabelle's approximation to a module system) helps to gain insight into the structure of an otherwise very large proof, by ensuring a separation of concerns. For example, the proof of $[\![\mathsf{T}_0(e)]\!] = [\![e]\!]$ has only the conditions from Definition 2 available, which shows that the cardinality analysis is irrelevant for functional correctness.

6.1 The formalization gap

Every formalization – whether hand-written or machine-checked – has a formalization gap, i.e. a difference to the formalized artifact that is not (and often cannot) be formally bridged. Despite the effort that went into this formalization, the gap is not very narrow, and in at least one instance has been wide enough to fall into:

- Clearly, we have not formalized the algorithm as implemented in GHC, but rather a mathematical description of it. Haskell code has no primitive function yielding a least fixed point, but has to find it using fixed-point iteration. Termination of the algorithm is not covered here.
- Our syntax is a much restricted variant of GHC's intermediate language Core. The latter is said to be simple, having just 15 constructors, but that is still a sizable chunk for a machine-checked formalization. Our meta-argument is that, for this particular theorem, our smaller syntax is representative.
- GHC's Core is typed, while we work in an untyped setting. The analysis, as implemented in GHC, ignores the types, so we argue that this is warranted. The general-purpose eta-expansion code used to implement the transformation will simply refrain from expanding a definition if its type does not obviously allow it, which can be the case with type functions. As the specifications used in the proofs require only lower bounds on the analysis results, the results still hold if one is more conservative than the analysis allows.
- In GHC, terms are part of modules and packages; this complexity is completely ignored here. The real implementation will, for example, not collect arity and co-call information for external identifiers, as they cannot be used anyway. This implementation short-cut is ignored here.
- Identifiers in GHC's core are annotated with a wealth of additional information – inlining information, occurrence information, strictness signatures, demand information. As later phases rely on these information, they have to be considered part of the language, and should be included in a formal semantics.

  This actually caused a nasty bug[1] that appeared in the third release candidate of GHC 7.10. The symptoms were weird: The program would skip over a call to error and simply carry on with the rest of the code. With Call Arity disabled, nothing bad happened. What went wrong?

  It boiled down to a function
  $f :: a \to b$
  $f\ x = error\ \text{“}\ldots\text{”}$
  which the strictness analyzer annotates with `<B,A>b`, indicating that once f is called with one argument, the result is definitely bottom.

  In the code at hand, every call to f passes two arguments, i.e. **case** $f\ x\ y$ **of** $\{\ldots\}$. Therefore Call Arity determines f's external arity to be 2, and changes the definition to
  $f\ x\ y = error\ \text{“}\ldots\text{”}\ y$

  The strictness annotation on f, however, is still present, allowing the simplifier to change the code that contains the call to f to **case** $f\ x$ **of** $\{\}$, as passing one argument is enough to cause the exception to be raised. It also removes all alternatives from the **case**, as the control flow will not return.

  On their own, each transformation is correct; together, havoc is created: Due to the eta-expansion, the evaluation of $f\ x$ does *not* raise an exception. Because the **case** expression has no alternatives any more, the control flow in the final program continues at some other, undefined part of the program.

  One way to fix this would be to completely remove annotations that might no longer be true after eta-expanding a function definition, losing the benefit that these annotations provide. The actual fix was more careful and capped the reported arity at the number of arguments with which, according to the strictness signature, the function is definitely bottom.
- There is no official semantics of GHC Core that is precise enough to observe sharing. The closest thing is Richard Eisenberg's work on formalizing Core [8], which includes a small step operational semantics for all of Core, but with call-by-name semantics. So the only "real" specification would be GHC's implementation, including all later stages and the runtime system, which is not a usable definition.
- Finally, our formal notion of performance is an approximation for real performance. Formally capturing the actual runtime of a program on modern hardware with multiple cores and complex caches is currently way out of reach.

[1] https://ghc.haskell.org/trac/ghc/ticket/10176

7. Related work

This work connects arity and cardinality analyses with operational safety properties, using an interactive theorem prover; as such this is a first.

However, this is not the first compiler transformation proven correct in an interactive theorem prover. After all there is CompCert (e.g. [17]), a complete verified optimizing compiler for C implemented in Coq. Furthermore, a verified Java to Java bytecode compiler [18] was written using Isabelle's code generation facilities, and the CakeML project has produced, among other things, a verified compiler from CakeML to CakeML bytecode, implemented in the HOL4 theorem prover [15]. Their theorems cover functional correctness of the compilers, though, but not performance.

Using a resource aware program logic for a subset of Java bytecode, which they have implemented in Isabelle, Aspinall, Beringer and Momigliano validate local optimizations [1] to be indeed optimizations with regard to a variety of resource algebras. The Isabelle formalizations of the proofs seem to be lost.

In the realm of functional programming languages, a number of formal treatments of compiler transformations exist, e.g. verification of the CPS transformation in Coq (e.g. [6], [7]), Twelf (e.g. [26]) or Isabelle (e.g. [20]). As their focus lies on finding proper techniques for handling naming, their semantics do not express heap usage and sharing.

Sand's *improvement theory* [23] provides an general, inequational algebra to describe the effect of program transformations on performance. Its notion of improvement is similar to our notion of safety, while the more general notion of weak improvement allows performance regressions up to a constant factor. This theory was adapted for lazy languages, both for improvement of time [21] and space [11, 12].

Recently, Hackett and Hutten [13] took up on Sands' work and built a general framework to prove *worker/wrapper transformations* time improving. And while neither that nor Sands's work have yet been machine-checked, at least the semantic correctness of Hutton's worker/wrapper framework has been verified using Isabelle [9].

Could we have built our results on theirs, especially as [13] uses almost the same abstract machine? Indeed, eta-expansion can be phrased as an instance of the worker/wrapper transformation, with abstraction and representation contexts $\mathsf{Abs} = []$ and $\mathsf{Rep} = (\lambda z_1 \ldots z_n. ([]\ z_1 \ldots\ z_n))$. Unfortunately, the assumptions of the worker/wrapper improvement theorem are not satisfied, and this is to be expected: Sands' notion of improvement – and hence Hackett and Hutton's theorems – guarantee improvement in *all* contexts, while in our case the eta-expansion is justified by an analysis of the actual context, and is generally unsafe in other contexts.

So in the current form, improvement theory is tailored to local transformations and, as Sands points out in [12], would require the introduction of context information to apply to whole-program transformations such as Call Arity. Such a grand unified improvement theory for call-by-need would be a tremendously useful thing to have.

Related to the Call Arity analysis are the GHC's "regular" arity analysis, which is described in working notes by Xu and Peyton Jones [28], and its cardinality analysis, most recently described in [24]. See [5] for a detailed discussion.

8. Conclusion

First and foremost, we have proven that Call Arity is a safe transformation.

That was initially not the case: Working towards a formally precise understanding of Call Arity uncovered a bug in the implementation, where thunks would erroneously be eta-expand when they are part of a linearly recursive binding.[2] So the work was useful. But that alone does not warrant the effort put into this work – this bug would have been spotted by someone eventually, and indeed the formalization gap is still wide enough to hide bugs from our formal tools.

What made this work worth it is the scarcity of formal treatments of the performance effects of compiler transformations, so it is an additional data point to the question "How practical is it, yet?". Our answer here is, yes, it is possible, but still too tedious, and the formalization gap is a bit too wide.

We have created reusable artifacts – syntax, semantics, data structures – that make similar endeavors, e.g. a safety proof of the cardinality analysis described in [24], more tractable.

It would be very desirable to narrow the formalization gap and formalize GHC's Core in Isabelle. Using Isabelle's code generation to Haskell, even verified implementations of Core-to-Core transformations in GHC appear possible. This would be a milestone on the way to formally verified compilation of Real-World-Haskell.

Acknowledgments

This work was supported by the Deutsche Telekom Stiftung. I'd like to thank the KIT's Programming Paradigms Group and Rebecca Schwerdt for helpful proofreading of the paper.

References

[1] D. Aspinall, L. Beringer, and A. Momigliano. Optimisation validation. In *COCV'06*, volume 176(3) of *ENTCS*, pages 37 – 59, 2007.

[2] C. Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014.

[3] J. Breitner. The correctness of Launchbury's natural semantics for lazy evaluation. *Archive of Formal Proofs*, Jan. 2013. http://afp.sf.net/entries/Launchbury.shtml.

[4] J. Breitner. The Safety of Call Arity. *Archive of Formal Proofs*, Feb. 2015. http://afp.sf.net/entries/Call_Arity.shtml.

[5] J. Breitner. Call Arity. In *TFP'14*, volume 8843 of *LNCS*, pages 34–50. Springer, 2015.

[6] A. Chlipala. A verified compiler for an impure functional language. In *POPL'10*, pages 93–106. ACM, 2010.

[7] Z. Dargaye and X. Leroy. Mechanized Verification of CPS Transformations. In *LPAR'07*, volume 4790 of *LNCS*, pages 211–225. Springer, 2007.

[8] R. Eisenberg. System FC, as implemented in GHC, 2013. URL https://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf.

[9] P. Gammie. The worker/wrapper transformation. *Archive of Formal Proofs*, Oct. 2009. http://afp.sf.net/entries/WorkerWrapper.shtml.

[10] A. J. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, 1996.

[11] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In *HOOTS'99*, volume 26 of *ENTCS*, pages 69–86, 1999.

[12] J. Gustavsson and D. Sands. Possibilities and limitations of call-by-need space improvement. In *ICFP'01*, pages 265–276. ACM, 2001.

[13] J. Hackett and G. Hutton. Worker/Wrapper/Makes It/Faster. In *ICFP'14*, pages 95–107. ACM, 2014.

[14] B. Huffman. *HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs*. PhD thesis, Portland State University, 2012.

[15] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: A verified implementation of ml. In *POPL'14*, pages 179–191. ACM, 2014.

[16] J. Launchbury. A natural semantics for lazy evaluation. In *POPL*, 1993.

[17] X. Leroy. Mechanized semantics for compiler verification. In *APLAS'12*, volume 7705 of *LNCS*, pages 386–388. Springer, 2012. Invited talk.

[18] A. Lochbihler. Verifying a compiler for java threads. In *ESOP'10*, volume 6012 of *LNCS*, pages 427–447. Springer, Mar. 2010.

[19] S. Marlow and S. L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.

[20] Y. Minamide and K. Okuma. Verifying CPS Transformations in Isabelle/HOL. In *MERLIN'03*, pages 1–8. ACM, 2003.

[21] A. K. Moran and D. Sands. Improvement in a Lazy Context: An Operational Theory for Call-By-Need. In *POPL'99*, pages 43–56. ACM, 1999.

[22] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[23] D. Sands. Operational theories of improvement in functional languages (extended abstract). In *Glasgow Workshop on Functional Programming*, Workshops in Computing Series, pages 298–311. Springer, August 1991.

[24] I. Sergey, D. Vytiniotis, and S. Peyton Jones. Modular, Higher-order Cardinality Analysis in Theory and Practice. POPL, 2014.

[25] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7:231–264, 1997.

[26] Y. H. Tian. Mechanically Verifying Correctness of CPS Compilation. In *CATS'06*, volume 51 of *CRPIT*, pages 41–51. ACS, 2006.

[27] C. Urban and C. Kaliszyk. General Bindings and Alpha-Equivalence in Nominal Isabelle. *Logical Methods in Computer Science*, 8(2), 2012. .

[28] D. N. Xu and S. Peyton Jones. Arity analysis, 2005. Working Notes.

[2] see GHC commit 306d255