Submitted to ICFP '15

Formally Proving a Compiler Transformation Safe

Joachim Breitner
Karlsruhe Institute of Technology
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 when formally working with cardinality analyses.

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, which we call *safety*, 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.

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 Call Arity [4] analysis was recently added to GHC. It combines a fairly straight-forward 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 bound value is called, is required to eta-expand a thunk, as that is only safe if the thunk is called 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.

This would not be ruled out by simply proving correctness with regard to a standard denotational semantics! Instead, a more detailed semantics is required. We use a small-step operational semantics akin to Sestoft's mark 1 abstract machine, which models an explicit heap and allows us to prove that the number of heap allocations does not increase by transforming the program. We argue that this is a suitable criterion for safety.

Our contributions are:

- We prove that the Call Arity analysis is indeed 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, used in one of the refinements.
- 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 in 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. We discuss the development, including the required effort and the remaining formalization gap, in Sec. 6.

2. Overview and example

The remainder of the paper will be formal enough, so to give a better intuition and overview, we look at a small example before introducing the syntax, semantics, transformations and analyses more rigorously in the following 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 [4].

2.1 From the example...

Consider the following program, written in some pure lazy functional programming language with sharing, e.g. Haskell:

```
foo a = let t1 = f1 a in
        let t2 = f2 a in
        let g x = if p x else t1 in then g (x + t2 x)
        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*.

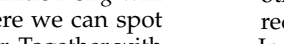
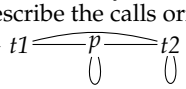
Functional programs run faster if a function like *g* expects as many parameters as possible [14]. We therefore want to eta-expand its definitions to match the number of arguments it is called with.

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

```
let g x y = (if p x else t1 then g (x + t2 x) ) y
```

We also see that both *t1* and *t2* are always called with one argument. Can we eta-expand their definition to **let** *t1* y = *f1 a y* resp. **let** *t2* y = *f2 a y*? It depends! If we eta-expand *t1* then the evaluation of *f1 a* will no longer be shared between multiple invocations of *t1*. As we do not know anything about *f1* we have to pessimistically assume this to be an expensive operation that we must not suddenly repeat. Therefore, we can only eta-expand *t1* if we can guarantee that it is called at most once.

That is why a good arity analyses needs the help of a precise cardinality analysis. For *t2*, a conservative analysis will tell us that it might be called multiple times by the recursive function *g*, so we would not eta-expand *t2*. For *t1*, a precise analysis might however be able to determine that it is called at most once, allowing us to eta-expand its definition. How could the 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. Analyzing the definition of *g* will yield the graph  where we can spot 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 whole **let** with the graph  where the absence of a loop at *t1* implies the desired cardinality information.

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 the general case?

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 very close 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 might 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 such work could always involve allocations. It is also realistic: When working on GHC, the number of bytes allocated by the benchmarks and test cases is the prime measure that is observed to detect improvements and regressions.

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, for two reasons:

- It is nice to separate various aspects of the proof (e.g. the interaction of the arity analysis with the cardinality analysis; the gap between the steps of the semantics and the structural 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 variables are calling each other in a nice, linear fashion among other, more complex recursion patterns.

In the first iteration, 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. We give conditions (Definition 6) when an arity analysis using this cardinality analysis is safe (Lemma 3). 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*.

The next iteration 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.

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

Figure 1. A simple lambda calculus

$$\frac{}{(\Gamma, e\ x, S) \Rightarrow (\Gamma, e, \$x{\cdot}S)}\ \text{APP}_1$$

$$\frac{}{(\Gamma, \lambda y.\, e, \$x{\cdot}S) \Rightarrow (\Gamma, e[y := x], S)}\ \text{APP}_2$$

$$\frac{(x \mapsto e) \in \Gamma}{(\Gamma, x, S) \Rightarrow (\Gamma \setminus x, e, \#x{\cdot}S)}\ \text{VAR}_1$$

$$\frac{\mathsf{isVal}(e)}{(\Gamma, e, \#x{\cdot}S) \Rightarrow (\Gamma[x \mapsto e], e, S)}\ \text{VAR}_2$$

$$\frac{}{(\Gamma, (e\,\mathbf{?}\,e_\mathbf{t} : e_\mathbf{f}), S) \Rightarrow (\Gamma, e, (e_\mathbf{t} : e_\mathbf{f}){\cdot}S)}\ \text{IF}_1$$

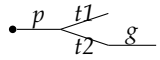
$$\frac{b \in \{\mathbf{t}, \mathbf{f}\}}{(\Gamma, \mathbf{C}_b, (e_\mathbf{t} : e_\mathbf{f}){\cdot}S) \Rightarrow (\Gamma, e_b, S)}\ \text{IF}_2$$

$$\frac{\mathsf{dom}\,\Delta \cap \mathsf{fv}(\Gamma, S) = \{\}}{(\Gamma, \mathbf{let}\ \Delta\ \mathbf{in}\ e, S) \Rightarrow (\Delta \cdot \Gamma, e, S)}\ \text{LETREC}$$

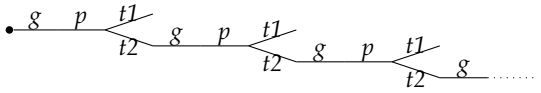
Figure 2. The operational semantics

Given such a trace tree analysis, an abstract analysis as described in the first iteration can be implemented: The trees describing the expressions of a configuration 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, the cardinalities of the individual variables can be determined. We specify conditions on the trace tree analysis (Definition 9) and show them to be sufficient to fulfill the specification of the first iteration (Lemma 4).

In our example, after g has been put on the heap, the tree corresponding to the definition of g, namely



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



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

The third and final iteration 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 to be sufficient (Lemma 5).

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

The following three quite 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 [3]; the proof document contains a table that maps the definitions and lemmas of this paper to the corresponding entities in the Isabelle development.

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, so we use booleans ($\mathbf{C_t}, \mathbf{C_f}$) with an if-then-else construct as representatives for data types and case expressions.

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 [11] and Sestoft [18], we require application arguments to be variables. This way, it is sufficient to model sharing for let-bound expressions.

We denote the set of free variables of an expression e 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 $\mathsf{isVal}(e)$ is true if e is a lambda abstraction or a constructor, and false otherwise.

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.

The proper treatment of names is the major technical hurdle when rigorously formalizing anything related to the lambda calculus. We employ Nominal Logic [20] here, so the lambda abstractions and 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 $\$x{\cdot}S$,
- update markers, written $\#x{\cdot}S$, and
- alternatives of conditionals, written $(e_1 : e_2){\cdot}S$.

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 [18], 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.

In the interest of naming hygiene, the names for the new bindings in the LETREC 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 technically not deterministic, as there is an infinite number of valid names that can be used when putting the bindings onto the heap. A nice consequence of this is that our proofs cannot make short-cuts using determinism, so adding "real" nondeterminism should not pose a problem.

Note that the semantics takes good configurations to good configurations.

This semantics is equivalent to Launchbury's natural semantics [11], which in turn is correct and adequate with regard to a standard denotational semantics; these proofs are machine-verified as well [2].

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 [21]. Here, we treat e as a black box and instead look 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 hence is:

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

We want $\alpha - 1$ to be defined everywhere, so here, $0 - 1 = 0$. By convention, $\bar{\alpha}$ is a partial map from variables to arities, and $\dot{\alpha}$ a list of arities.

4. Arity analyses

An arity analysis is thus a function that, given a binding (consisting of the bound values Γ and the body e), determines the arity of each of the bound values. It depends on the number α of arguments passed to e and may return $\bot$ for a bound value that is not called at all:

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

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:

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

The actual transformation happens at a binding, where we eta-expand bindings 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 when the transformed program does not perform more allocations than the original program. A – technical – benefit of this measure is that the number of allocations made 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 $|\operatorname{dom}\Gamma'| \leq |\operatorname{dom}\Gamma|$.

An arity analysis $\mathcal{A}$ is safe if the transformation based on it 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:

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

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)

$$\operatorname{dom}\mathcal{A}_\alpha(e) \subseteq \mathsf{fv}\, e \qquad \text{(A-dom)}$$

$$\operatorname{dom}\mathcal{A}_\alpha(\Gamma, e) \subseteq \operatorname{dom}\Gamma \qquad \text{(Ah-dom)}$$

$$z \notin \{x, y\} \implies \mathcal{A}_\alpha(e[x := y])\ z = \mathcal{A}_\alpha(e)\ z \qquad \text{(A-subst)}$$

$$x, y \notin \operatorname{dom}\Gamma \implies \mathcal{A}_\alpha(\Gamma[x := y], e[x := y]) = \mathcal{A}_\alpha(\Gamma, e) \qquad \text{(Ah-subst)}$$

$$[x \mapsto \alpha] \sqsubseteq \mathcal{A}_\alpha(x) \qquad \text{(A-Var)}$$

$$\mathcal{A}_{\alpha+1}(e) \sqcup [x \mapsto 0] \sqsubseteq \mathcal{A}_\alpha(e\ x) \qquad \text{(A-App)}$$

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

$$\mathcal{A}_0(e) \sqcup \mathcal{A}_\alpha(e_\mathbf{t}) \sqcup \mathcal{A}_\alpha(e_\mathbf{f}) \sqsubseteq \mathcal{A}_\alpha(e\ \mathbf{?}\ e_\mathbf{t} : e_\mathbf{f}) \qquad \text{(A-If)}$$

$$\overline{\mathcal{A}}_{\mathcal{A}_\alpha(\Gamma,e)}(\Gamma) \sqcup \mathcal{A}_\alpha(e) \sqsubseteq \mathcal{A}_\alpha(\Gamma, e) \sqcup \mathcal{A}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e) \qquad \text{(A-Let)}$$

where

$$\overline{\mathcal{A}}_{\bar{\alpha}}(\Gamma) := \bigsqcup \left\{ \mathcal{A}_{(\bar{\alpha}\ x)}(e) \middle| (x \mapsto e) \in \Gamma,\ \bar{\alpha}\ x \neq \bot \right\}. \qquad \square$$

These conditions come quite naturally: An expression should not report calls to variables that it does not know about. Replacing one variable by another should not affect the arity 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. We also assume that the analysis is not higher-order, e.g. for an expression $e\ x$, nothing useful is known on how x is called. This means that the following proofs needs modifications before they can applied to such an extended analysis.

In rule (A-If), the scrutinee is evaluated without arguments, hence 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 $\mathcal{A}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e)$ are disjoint, i.e. $\mathcal{A}_\alpha(\Gamma, e)$ contains the information on how the variables of the current binding are called, while $\mathcal{A}_\alpha(\mathbf{let}\ \Gamma\ \mathbf{in}\ e)$ 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 $\mathcal{A}_\alpha(\Gamma, e)$, which anticipates the fixed-point iteration in the implementation of the analysis.

These conditions are sufficient 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, the eta-expanded expression evaluates to the original expression:

Lemma 1 (Safety of eta-expansion)

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

PROOF

$$\begin{aligned}
&(\Gamma, \mathcal{E}_\alpha(e), \$x_1 \cdots \$x_\alpha \cdot S) \\
&= (\Gamma, (\lambda z_1 \ldots z_\alpha . e\ z_1 \ldots z_\alpha), \$x_1 \cdots \$x_\alpha \cdot S) \\
&\Rightarrow^* (\Gamma, e\ x_1 \ldots x_\alpha, S) && \{ \text{by } \textsc{app}_2 \} \\
&\Rightarrow^* (\Gamma, e, \$x_1 \cdots \$x_\alpha \cdot S) && \{ \text{by } \textsc{app}_1 \} \ \blacksquare
\end{aligned}$$

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

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

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

$\square$

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 $x \in \mathsf{thunks}\,\Gamma$. $\square$

We did not include this requirement in definition of $\triangleright$ as we want to use that later, 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. Note that the semantics will, upon evaluating a variable, always take the binding off the heap, even if it is already a value:

$$\begin{aligned}
(\Gamma[x \mapsto (\lambda y.\, y)], x, S) &\Rightarrow (\Gamma, (\lambda y.\, y), \#x\cdot S) \\
&\Rightarrow (\Gamma[x \mapsto (\lambda y.\, y)], (\lambda y.\, y), S)
\end{aligned}$$

We would not be able to prove consistency in the intermediate state. To work around this issue, assume that rule $\textsc{var}_1$ had an additional constraint $\neg\,\mathsf{isVal}(e)$ and that the rule

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

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 $\mathsf{T}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, e, S)) \Rightarrow^* \mathsf{T}_{(\bar\alpha',\alpha',\dot\alpha')}((\Gamma', e', S'))$. $\square$

PROOF by the individual steps of $\Rightarrow^*$.

For example, for $\textsc{app}_1$ 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 $\textsc{app}_1$.

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

In case $\textsc{var}_1$ 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 $\textsc{var}_2$.

The actual eta-expansion is treated is case $\textsc{var}'_1$: 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

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

Case $\textsc{let}_1$: The new variables in Δ are fresh with regard to Γ and S, hence also with regard to $\bar\alpha$ according to the naming hygiene conditions in $(\bar\alpha, \alpha, \dot\alpha) \triangleright_\mathbf{N} (\Gamma, \mathbf{let}\ \Delta\ \mathbf{in}\ e, S)$. So in order to have $(\mathcal{A}_\alpha(\Delta, e) \sqcup \bar\alpha, \alpha, \dot\alpha) \triangleright (\Delta \cdot \Gamma, e, S)$, it suffices show

$$(\overline{\mathcal{A}}_{\mathcal{A}_\alpha(\Delta,e)}(\Delta) \sqcup \mathcal{A}_\alpha(e))\big|_{\mathsf{dom}\,\Delta\cup\mathsf{dom}\,\Gamma\cup\#S} \sqsubseteq \mathcal{A}_\alpha(\Delta, e) \sqcup \bar\alpha,$$

which follows from (A-Let) and $\mathcal{A}_\alpha(\mathbf{let}\ \Delta\ \mathbf{in}\ e)\big|_{\mathsf{dom}\,\Gamma\cup\#S} \sqsubseteq \bar\alpha$. The requirement $\mathcal{A}_\alpha(\Delta, e)\ x = 0$ for $x \in \mathsf{thunks}\,\Delta$ holds by (Ah-thunk). $\blacksquare$

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 (in the sense of Definition 1): 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 $(\bot, 0, []) \rhd_{\mathbf{N}} ([], e, [])$, so Lemma 2 gives us $\bar{\alpha}$, α and $\dot{\alpha}$ so that $\mathsf{T}_{(\bot,0,[])}(([], e, [])) \Rightarrow^* \mathsf{T}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, v, []))$. The corollary holds because Γ and $\overline{\mathsf{T}}_{\bar{\alpha}}(\Gamma)$ bind the same variables, and $\dot{\mathsf{T}}_{\dot{\alpha}}([]) = []$. ■

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

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

and

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

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 [8].

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 determines how often a bound value 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 "no 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:

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

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

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

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

Furthermore, the cardinality analysis is not higher order and hence 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.

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

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

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

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

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

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

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

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

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

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

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

□

Safety The safety proof proceeds similarly to the one in 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:

- In the transformed program, the eta-expanded expression is now a value, so 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, it can interfere with uses of Lemma 1.
- Because the eta-expanded expression is now 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 the update markers $\#x$ where $x \in r$.

We also keep track of $\bar\gamma\colon \text{Var} \to \mathsf{Card}$, which remembers the current cardinality of the variables in the configuration:

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.

Case LET_1: Besides the usual calculations about scoping and freshness, we use the specification rules (C-Let) and (Ah-∞-thunk) to ensure that $(\Delta \cdot \Gamma, e, s) \rhd_{\mathbf{c}} (\mathcal{A}_\alpha(\Delta, e) \sqcup \bar\alpha, \alpha, \dot\alpha, \mathcal{C}_\alpha(\Delta, e) \sqcup \bar\gamma, r)$ holds. ■

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 somewhat 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 ($\text{Var} \to \mathbb{N}_\bot$) 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 $e\,?\,x : y$).

So we need a richer type that captures the future calls of an expression, can distinguish different code paths and maps easily to $\text{Var} \to \mathbb{N}_\bot$: 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 of an expression (or a whole configuration), 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 set in the list, its prefixes are also in the set). Each such list corresponds to a (finite) path in the tree. The function $\mathsf{paths}\colon \mathrm{TTree} \to 2^{[\mathrm{Var}]}$ implements this correspondence.

Another view is given by the function

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

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. The tree with exactly one edge labeled x is simply written x. The tree $t \setminus V$ is t with all edges with labels in V contracted, $t\big|_V$ 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 $s\colon (\mathrm{Var} \to \mathrm{TTree}) \to \mathrm{TTree} \to \mathrm{TTree}$ defined by

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

that traverses the tree and upon every call interleaves the tree of the called expressions with the remainder of the current tree.

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

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

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

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 $(\mathrm{Var} \to \mathbb{N}_\bot)$, as required for a cardinality analysis, using $c\colon \mathrm{TTree} \to (\mathrm{Var} \to \mathbb{N}_\bot)$ defined by

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

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. As the resulting value might be passed to unknown code or stored in a data structure, we cannot assume anything about how often the resulting value is used. This justifies the arity parameter: We expect $\mathcal{T}_0(\lambda x.\, y) = y^*$ but $\mathcal{T}_1(\lambda x.\, y) = y$.

We write $\overline{\mathcal{T}}_{\bar{\alpha}}(\Gamma)$ 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 what 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:

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

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

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

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

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

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

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):

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

And finally, the connection to the arity analysis:

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

□

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:

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

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:

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

With this we can define

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

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 conditions (C-dom) and (Ch-dom) follow directly from (T-dom) and (Th-dom) with (A-dom) and (Ah-dom).

The conditions (C-cong), (C-not-called) and (C-upd) follow directly from the definitions of $\overline{\mathcal{T}}$ and $\dot{\mathcal{T}}$

We have $x^* \sqsubseteq \dot{\mathcal{T}}_{\dot\alpha}(S)$ for $\$x \in S$, so (C-args) follows from $[x \mapsto \infty] = c(x^*) \sqsubseteq c(\dot{\mathcal{T}}_{\dot\alpha}(S)) \sqsubseteq (\mathcal{C}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, e, S)))$. Similar calculations prove (C-call) using (T-Var), (C-App) using (T-App), (C-Lam) using (T-subst) and (T-Lam), (C-If$_1$) using (T-If).

Condition (C-If$_2$) is where the precision comes from, as we retain the knowledge that two code paths are mutually exclusive. The proof is a direct consequence of $t \sqsubseteq t \oplus t'$.

The variable cases are interesting, as these interact with the heap, and hence with the s function.

We first show that (C-Var$'_1$) is fulfilled. Abbreviate $T \coloneqq \mathsf{thunks}\,\Gamma$ and note that $x \notin T$. We have

$$\begin{aligned}
&\mathcal{C}_{(\bar\alpha,\bar\alpha\ x,\dot\alpha)}((\Gamma, e, \#x{\cdot}S)) \\
&= c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (\mathcal{T}_{\bar\alpha\ x}(e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \\
&= c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (\mathsf{next}\ x\ x \otimes \mathcal{T}_{\bar\alpha\ x}(e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \\
&\qquad\qquad \{\text{ as } \mathsf{next}\ x\ x = \bullet \} \\
&\sqsubseteq c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (\mathsf{next}\ x\ (x \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))) \otimes \mathcal{T}_{\bar\alpha\ x}(e))\big) \\
&\qquad\qquad \{\text{ using } (\mathsf{next}\ x\ t) \otimes t' \sqsubseteq \mathsf{next}\ x\ (t \otimes t') \} \\
&= c\big(\ \mathsf{next}\ x\ (s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (x \otimes \dot{\mathcal{T}}_{\dot\alpha}(S)))\big) \\
&\qquad\qquad \{\text{ by the definition of } s \} \\
&\sqsubseteq c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (x \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) - x \\
&\sqsubseteq c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (\mathcal{T}_{\alpha}(x) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) - x \qquad \{\text{ by (T-Var) } \} \\
&= \mathcal{C}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, x, S)) - x.
\end{aligned}$$

Condition (C-Var$_1$) represents the evaluation of a thunk. The proof is analog, using $\overline{\mathcal{T}}_{\bar\alpha}(\Gamma)[x \mapsto \bullet] = \overline{\mathcal{T}}_{\bar\alpha}(\Gamma')$ in the step where the definition of s is unfolded.

For (C-Var$_2$) abbreviate $T \coloneqq \mathsf{thunks}\,\Gamma = \mathsf{thunks}(\Gamma[x \mapsto e])$. We know $\mathsf{isVal}(e)$, so $\mathcal{T}_0(e)$ is repeatable, by (T-value). 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:

$$s_T\ (\bar t[x \mapsto t])\ (t \otimes t') = s_T\ \bar t\ (t \otimes t')$$

Altogether, we show

$$\begin{aligned}
&\mathcal{C}_{(\bar\alpha,0,\dot\alpha)}((\Gamma[x \mapsto e], e, S)) \\
&= c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma[x \mapsto e]))\ (\mathcal{T}_0(e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \\
&= c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma)[x \mapsto \mathcal{T}_{\bar\alpha\ x}(e)])\ (\mathcal{T}_0(e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \\
&\sqsubseteq c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma)[x \mapsto \mathcal{T}_0(e)])\ (\mathcal{T}_0(e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \\
&= c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (\mathcal{T}_0(e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \quad \{\text{ by } \mathcal{T}_0(e) \text{ repeatable } \} \\
&= \mathcal{C}_{(\bar\alpha,0,\dot\alpha)}((\Gamma, e, \#x{\cdot}S)).
\end{aligned}$$

Proving condition (C-Let) is for the most part a tedious calculation involving freshness of variables. We use that if the domain of $\bar t'$ is disjoint from the variables occurring in $\bar t$ (i.e. $\forall y.\, \forall x \in \bar t\ y.\, \bar t'\ x = \bullet$), then

$$s_T\ (\bar t \sqcup \bar t')\ t = s_T\ \bar t\ (s_T\ \bar t'\ t).$$

Abbreviating $T \coloneqq \mathsf{thunks}\,\Gamma$ and $T' \coloneqq \mathsf{thunks}\,\Delta$, we show:

$$\begin{aligned}
&\mathcal{C}_{(\mathcal{A}_\alpha(\Delta,e)\sqcup\bar\alpha,\alpha,\dot\alpha)}((\Delta \cdot \Gamma, e, S)) \\
&= c\big(s_{T\cup T'}\ (\overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Delta,e)\sqcup\bar\alpha}(\Gamma \cdot \Delta))\ (\mathcal{T}_\alpha(e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \\
&= c\big(s_{T\cup T'}\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (s_{T\cup T'}\ (\overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Delta,e)}(\Delta))\ (\mathcal{T}_\alpha(e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S)))\big) \\
&\qquad\qquad \{\text{ by the above equation } \} \\
&= c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (s_{T'}\ (\overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Delta,e)}(\Delta))\ (\mathcal{T}_\alpha(e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S)))\big) \\
&= c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (s_{T'}\ (\overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Delta,e)}(\Delta))\ (\mathcal{T}_\alpha(e)) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \\
&\qquad\qquad \{\text{ as } \mathsf{dom}\,\Delta \text{ is fresh with regard to } S \} \\
&= c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (s_{T'}\ (\overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Delta,e)}(\Delta))\ (\mathcal{T}_\alpha(e)) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big)\big|_{\mathsf{dom}\,\Delta} \sqcup \\
&\quad\ c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (s_{T'}\ (\overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Delta,e)}(\Delta))\ (\mathcal{T}_\alpha(e)) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \setminus \mathsf{dom}\,\Delta \\
&= c\big(s_{T'}\ (\overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Delta,e)}(\Delta))\ (\mathcal{T}_\alpha(e))\big|_{\mathsf{dom}\,\Delta}\big) \sqcup \\
&\quad\ c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (s_{T'}\ (\overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Delta,e)}(\Delta))\ (\mathcal{T}_\alpha(e)) \setminus \mathsf{dom}\,\Delta \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \\
&\sqsubseteq c\big(\mathcal{T}_\alpha(\Delta, e)\big) \sqcup c\big(s_T\ (\overline{\mathcal{T}}_{\bar\alpha}(\Gamma))\ (\mathcal{T}_\alpha(\mathsf{let}\ \Delta\ \mathsf{in}\ e) \otimes \dot{\mathcal{T}}_{\dot\alpha}(S))\big) \\
&\qquad\qquad \{\text{ by (Th-s) and (T-Let) } \} \\
&= \mathcal{C}_\alpha(\Delta, e) \sqcup \mathcal{C}_{(\bar\alpha,\alpha,\dot\alpha)}((\Gamma, \mathsf{let}\ \Delta\ \mathsf{in}\ e, S)).
\end{aligned}$$

Finally, (Ah-∞-thunk) follows directly from (Th-∞-thunk). ■

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 on the set of variables with loops. 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.

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

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 $V^2 \coloneqq V \times V$. The set of neighbors of a variable is $N_x(G) \coloneqq \{y \mid x\text{—}y \in G\}$. 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 co-call graph to a TTree using the function $t\colon \mathrm{Graph} \to \mathrm{TTree}$, defined via

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

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

$$g(t) \coloneqq \bigcup\{\dot g(\dot x) \mid \dot x \in \mathsf{paths}\,t\}$$

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

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

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

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

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

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

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

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

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 bound values,

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

and finally the extra edges between what is called from a bound value and whatever the bound value is called with:

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

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

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

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:

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

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

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

□

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:

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

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 $S = \mathsf{dom}\,\Gamma$.

Condition (Th-s) is trivially true in the case of a recursive binding. For a non-recursive Γ, it follows from $(s_{\mathsf{thunks}\,\Gamma}\ (\overline{\mathcal{T}}_{\mathcal{A}_\alpha(\Gamma,e)}(\Gamma))\ (\mathcal{T}_\alpha(e)))\big|_{\mathsf{dom}\,\Gamma} = \mathcal{T}_\alpha(e)\big|_{\mathsf{dom}\,\Gamma} = \mathcal{T}_\alpha(\Gamma, e)$.

Finally, (Th-∞-thunk) is a direct consequence of (Rec-∞-thunk) and (Nonrec-∞-thunk). ■

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.

We first give the arity analysis:

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

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 $\mathcal{A}_\alpha(\Gamma, e) := \bar{\alpha}\big|_{\mathsf{dom}\,\Gamma}$ 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

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

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

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

which calculates the co-calls of an individual binding, adding the extra edges between multiple invocations of a bound value, unless it is 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 [16]. 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 [3]. 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 ([20], implemented for Isabelle in Christian Urban's Nominal2 package) helped a lot here, but still incurs technical overhead, as all involved definitions have to be proven equivariant, i.e. oblivious to variable renaming. While usually simple to prove, these 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 used are actually well-defined. Here, the HOLCF package by Brian Huffman [10] 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* ([1], 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.

Another benefit of having a machine-checked proof is that one can be a bit more liberal in the write-up for human consumption, i.e. this paper, and skip over uninteresting technical details with a much better conscience, knowing there really is nothing lurking in the skipped parts. We hope that this made the paper a bit more pleasant to read than if it were completely rigorous.

6.1 The formalization gap

Every formalization has a formalization gap, i.e. a difference to the formalized artifact that is not (and often cannot) be machine-checked. Despite the effort that went into this formalization, the gap is not very narrow:

- 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, and involves user-defined data types and arbitrary expressions as arguments to an application. 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. As the analysis in GHC ignores the types, we argue that this is warranted, but again, this leaves a small gap.
- In GHC, terms are part of modules and packages; this complexity is completely ignored here. The real implementation will, for example, not report arity information for external identifiers, as they cannot be used anyway. This implementation short-cut is ignored here.
- 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 [7], 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, at best, an approximation for real performance. Formally capturing

the actual runtime of a program on modern hardware with multiple cores and complex caches is currently impossible.

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. [12]), a complete verified optimizing compiler for C implemented in Coq. Furthermore, a verified Java to Java bytecode compiler [13] was written using Isabelle's code generation facilities. Their theorems cover functional correctness of the compilers, though, but not performance.

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. [5], [6]), Twelf (e.g. [19]) or Isabelle (e.g. [15]). As their focus lies on finding proper techniques for handling naming, their semantics do not express heap usage and sharing.

Narrowing it down to treatments of functional programs with lazy evaluation which focus on program transformations and their effect on performance, there is Sands' improvement theory (e.g. [9]), which aims to provide a foundation for proving transformations safe with regard to operational properties. His notion of strong improvement is similar to our notion of safety, but quantifies over all contexts. As such, it is more suited for analyzing local transformations than context-aware analyses like Call Arity.

Related to the Call Arity analysis are the GHC's "regular" arity analysis, which is described in working notes by Xu and Peyton Jones [21], and its cardinality analysis, most recently described in [17]. See [4] for a discussion of their relation to Call Arity.

8. Conclusion

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

That was initially not the case: Only when we worked towards a formally precise understanding of Call Arity we uncovered a bug in the implementation, where thunks would erroneously be eta-expand when they are part of a linearly recursive binding.[1] So the work was useful. But that alone does not warrant the effort put into this work – the bug would have been spotted by someone eventually, and no airplane control program relies on the safety of this the analysis (we hope).

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 answer the question "How practical is it, yet?". Our answer here is, yes, it is possible, but still too tedious.

We have created reusable artifacts – syntax, semantics, data structures – that make similar endeavors, e.g. a safety proof of the cardinality analysis described in [17], more tractable.

It would be 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.

[1] see GHC commit 306d255

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

[2] 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.

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

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

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

[6] Z. Dargaye and X. Leroy. Mechanized verification of cps transformations. In *LPAR'07*, volume 4790 of *LNCS*, pages 211–225. Springer, 2007.

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

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

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

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

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

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

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

[14] 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.

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

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

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

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

[19] Y. H. Tian. Mechanically verifying correctness of cps compilation. In *CATS'06*, volume 51 of *CRPIT*, pages 41–51. ACS, 2006.

[20] C. Urban and C. Kaliszyk. General bindings and alpha-equivalence in nominal isabelle. *Logical Methods in Computer Science*, 8(2), 2012. .

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