Thesis Proposal: Analysis and Transformation of Intrinsically Typed Syntax

Matthias Heinzel

December 22, 2022

Contents

1 Introduction **2**

2 Background **3**
- 2.1 Program Analysis and Transformation 3
- 2.2 Binding Representation 4
- 2.3 Intrinsically Typed Syntax 5

3 Preliminary Results **6**
- 3.1 Dead Binding Elimination 6
- 3.2 λ-calculus with Let-bindings 8

4 Timetable and Planning **9**
- 4.1 Further Work 9
- 4.2 Schedule 13

References **15**

A Ethics Quick Scan **17**

1 Introduction

When writing a compiler for a programming language, an important consideration is the treatment of binders and variables. A well-known technique when using dependently typed programming languages such as Agda [16] is to define an intrinsically typed syntax tree, where expressions are scope- and type-correct by construction and admit a total evaluation function [2]. This construction has featured in several papers, exploring basic operations like renaming and substitution [1] as well as compilation to different target languages [22, online supplementary material].

At the same time, there are large classes of important transformations that have not yet received much attention in an intrinsically typed setting. Optimisations, for example, play a central role in practical compilers and establishing their correctness is often not trivial, with ample opportunity for binding-related mistakes [24] [13]. Letting the type checker keep track of important invariants promises to remove common sources of bugs. A mechanised proof of semantics preservation can increase confidence further.

In return for the correctness guarantees, some additional work is required. Program *analysis* not only needs to identify optimisation opportunities, but potentially also provide a proof witness that the optimisation is safe, e.g. that some dead code is indeed unused. For the *transformation* of the intrinsically typed program, the programmer then has to convince the type checker that type- and scope-correctness invariants are preserved, which can be cumbersome. The goal of this thesis is to understand these consequences better and make the following contributions:

1. collect and document program analyses and transformations that can be performed on simple expression languages with variable binders
2. implement several of these transformations using intrinsically typed expressions in the dependently-typed programming language Agda
3. provide machine-checked proofs of the correctness (preservation of semantics) of the implemented transformations
4. attempt to apply relevant techniques from the literature, such as datatype-generic programming on syntax trees
5. identify common patterns and try capturing them as reusable building blocks (e.g. as datatype-generic constructions)

The Ethics and Privacy Quick Scan of the Utrecht University Research Institute of Information and Computing Sciences was conducted (see Annex A). It classified this research as low-risk with no fuller ethics review or privacy assessment required.

2 Background

As a running example, we will consider a simple expression language with let-bindings, variables, primitive values (integers and Booleans), and a few binary operators. Since the transformations in this thesis primarily relate to variables and binders, the choice of possible values and additional primitive operations on them is mostly arbitrary. Extending the language with further values and operators is trivial.

$$\begin{aligned} P, Q ::=\ & v \\ \mid\ & P + Q \\ \mid\ & \textbf{let } x = P \textbf{ in } Q \\ \mid\ & x \end{aligned}$$

An expression can be bound to a variable x using a **let**-binding. Note that this makes the language equivalent to a restricted version of the simply typed λ-calculus, where λ-abstraction and application can only occur together as $(\lambda x.Q)\ P$. Encapsulating this pattern as $\textbf{let } x = P \textbf{ in } Q$ simplifies parts of the analysis and avoids the need for allowing functions as values.

2.1 Program Analysis and Transformation

For now we mainly consider transformations aimed at optimising functional programs. A large number of program analyses and optimisations are presented in the literature [15] [23] and used in production compilers such as the Glorious Haskell Compiler (GHC). We generally focus on those transformations that deal with variable binders, such as *inlining*, *let-floating*, *common subexpression elimination* and *dead binding elimination*, which is explained below.

Dead binding elimination An expression is not forced to make use of the whole context to which it has access. Specifically, a let-binding introduces a new variable, but it might never be used in the body. Consider for example the following expression:

$$\begin{aligned} &\textbf{let } x = 42 \textbf{ in} \\ &\quad \textbf{let } y = x + 6 \textbf{ in} \\ &\quad\quad \textbf{let } z = y + 7 \textbf{ in} \\ &\quad\quad\quad x \end{aligned}$$

Here, the binding for z is clearly unused, as the variable never occurs in the body. Such dead bindings can be identified by *live variable analysis* and consequently be removed.

Note that y is not needed either: Removing z will make y unused. Therefore, multiple iterations of live variable analysis and binding elimination might be required. Alternatively, *strongly live variable analysis* can achieve the same result in a single pass by only considering variables to be live if they are used in declarations of variables that are live themselves.

2.2 Binding Representation

Explicit names The syntax specified above treats variables as letters, or more generally strings, and one can use the same representation inside a compiler. While this is how humans usually write programs, it comes with several downsides. For example, some extra work is necessary if we want the equality of expressions to be independent of the specific variable names chosen (α-*equivalence*). Also, there are pitfalls like variable shadowing and variable capture during substitution, requiring the careful application of variable renamings [3].

There have been multiple approaches to help compiler writers maintain the relevant invariants, such as GHC's rapier [18], but these are generally still error-prone [13].

de Bruijn indices With *de Bruijn indices* [11], each variable is instead represented as a natural number, counting the number of nested bindings between variable occurence and its binding: $\langle 0 \rangle$ refers to the innermost binding, $\langle 1 \rangle$ to the next-innermost etc. If we adapt the syntax for let-bindings to omit the unnecessary variable name, the example expression from dead binding elimination is represented as follows:

$$\begin{array}{l}
\textbf{let}\ 42\ \textbf{in} \\
\quad \textbf{let}\ \langle 0 \rangle + 6\ \textbf{in} \\
\quad\quad \textbf{let}\ \langle 0 \rangle + 7\ \textbf{in} \\
\quad\quad\quad \langle 2 \rangle
\end{array}$$

This makes α-equivalence of expressions trivial and avoids variable capture, but there are still opportunities for mistakes during transformations. Adding or removing a binding requires us to add or subtract 1 from all free variables in the binding's body. We can see this in our example when removing the innermost (unused) let-binding:

$$\begin{array}{l}
\textbf{let}\ 42\ \textbf{in} \\
\quad \textbf{let}\ \langle 0 \rangle + 6\ \textbf{in} \\
\quad\quad \langle 1 \rangle
\end{array}$$

Other representations There are many other techniques[1] such as higher-order abstract syntax [21] and also combinations of multiple techniques, e.g. the locally nameless representation [7].

[1]There is an introductory blogpost [8] comparing options available in Agda.

2.3 Intrinsically Typed Syntax

Whether we use explicit names or de Bruijn indices, the language as seen so far makes it possible to represent expressions that are ill-typed (e.g. adding Booleans) or accidentally open (containing free variables). Evaluating such an expression leads to a runtime error; the evaluation function is partial.

When implementing a compiler in a dependently typed programming language, we can use de Bruijn indices to define *intrinsically typed syntax trees*, where type- and scope-correctness invariants are specified on the type level and verified by the type checker. This makes the evaluation function total [2]. Similarly, transformations on the syntax tree need to preserve the invariants. While the semantics of the expression could still change, guaranteeing type- and scope-correctness rules out a large class of mistakes. We will demonstrate the approach in Agda and start by defining the types that expressions can have.

```
data U : Set where
  BOOL : U
  NAT  : U
⟦_⟧ : U → Set
⟦ BOOL ⟧ = Bool
⟦ NAT ⟧  = Nat
```

To know if a variable occurence is valid, one must consider its *context*, the bindings that are in scope. With de Bruijn indices in an untyped setting, it would suffice to know the number of bindings in scope. In a typed setting, it is also necessary to know the type of each binding, so we represent the context by a list of types: One for each binding in scope, from innermost to outermost.

```
Ctx = List U
variable
  Γ : Ctx
  σ τ : U
```

During evaluation, each variable in scope has a value. Together, these are called an *environment* in a given context.

```
data Env : Ctx → Set where
  Nil  : Env []
  Cons : ⟦ σ ⟧ → Env Γ → Env (σ :: Γ)
```

A variable then is an index into its context, also guaranteeing that its type matches that of the binding. Since variable Ref σ Γ acts as a proof that the environment Env Γ contains an element of type σ, variable lookup is total.

```
data Ref (σ : U) : Ctx → Set where
  Top : Ref σ (σ :: Γ)
  Pop : Ref σ Γ → Ref σ (τ :: Γ)
```

```
lookup : Ref σ Γ → Env Γ → ⟦ σ ⟧
lookup Top     (Cons v env) = v
lookup (Pop i) (Cons v env) = lookup i env
```

Now follows the definition of intrinsically typed expressions, where an Expr is indexed by both its type (σ : U) and context (Γ : Ctx). We can see how the context changes when introducing a new binding that is then available in the body of a Let.

```
data Expr (Γ : Ctx) : (τ : U) → Set where
  Val  : ⟦ σ ⟧ → Expr Γ σ
  Plus : Expr Γ NAT → Expr Γ NAT → Expr Γ NAT
  Let  : Expr Γ σ → Expr (τ :: Γ) τ → Expr Γ τ
  Var  : Ref σ Γ → Expr Γ σ
```

This allows the definition of a total evaluator using an environment that matches the expression's context.

```
eval : Expr Γ σ → Env Γ → ⟦ σ ⟧
eval (Val v)       env = v
eval (Plus e₁ e₂) env = eval e₁ env + eval e₂ env
eval (Let e₁ e₂)  env = eval e₂ (Cons (eval e₁ env) env)
eval (Var x)       env = lookup x env
```

3 Preliminary Results

As a first step, we implemented one optimisation in Agda, including a mechanised proof of its preservation of semantics. The main ideas are outlined below; the full source code is available online[2].

3.1 Dead Binding Elimination

Sub-contexts To reason about the part of a context that is live (actually used), we introduce *sub-contexts*. Conceptually, these are contexts that admit an *order-preserving embedding* (OPE) [6] into the original context, and we capture this notion in a single data type. For each element of a context, a sub-context specifies whether to Keep or Drop it.

```
data SubCtx : Ctx → Set where
  Empty : SubCtx []
  Drop  : SubCtx Γ → SubCtx (τ :: Γ)
  Keep  : SubCtx Γ → SubCtx (τ :: Γ)
```

The context uniquely described by a sub-context is then given by a function ⌊_⌋ : SubCtx Γ → Ctx, and we further know its embedding.

[2] https://git.science.uu.nl/m.h.heinzel/correct-optimisations

We now define $\_\subseteq\_$: SubCtx Γ $\rightarrow$ SubCtx Γ $\rightarrow$ Set, stating that one sub-context is a subset of the other. Its witnesses are unique, which simplifies the correctness proofs. A similar relation on Ctx does not have this property (e.g. [NAT] can be embedded into [NAT, NAT] either by keeping the first element or the second), which would complicate equality proofs on terms including witnesses of $\_\subseteq\_$.

From now on, we will only consider expressions Expr $\lfloor \Delta \rfloor$ τ in some sub-context. Initially, we take Δ = all Γ : SubCtx Γ, the complete sub-context of the original context.

Analysis Now we can annotate each expression with its *live variables*, the sub-context $\Delta' \subseteq \Delta$ that is really used. To that end, we define annotated expressions LiveExpr Δ Δ' τ. While Δ is treated as Γ was before, Δ' now only contains live variables, starting with a singleton sub-context at the variable usage sites.

```
data LiveExpr : (Δ Δ′ : SubCtx Γ) (τ : U) → Set where
  Let : LiveExpr Δ Δ₁ σ →
    LiveExpr (Keep Δ) Δ₂ τ →
    LiveExpr Δ (Δ₂ ∪ pop Δ₂) τ
  ...
```

To create such annotated expressions, we need to perform some static analysis of our source programs. The function analyse computes an existentially qualified live sub-context Δ' together with a matching annotated expression. The only requirement we have for it is that we can forget the annotations again, with forget $\circ$ analyse $\equiv$ id.

```
analyse : Expr ⌊ Δ ⌋ τ → Σ[Δ′ ∈ SubCtx Γ] LiveExpr Δ Δ τ
forget : LiveExpr Δ Δ′ τ → Expr ⌊ Δ ⌋ τ
```

Transformation Note that we can evaluate LiveExpr directly, differing from eval mainly in the Let-case, where we match on Δ_2 to distinguish whether the bound variable is live. If it is not, we directly evaluate the body, ignoring the bound declaration. Another important detail is that evaluation works under any environment containing (at least) the live context.

```
evalLive : LiveExpr Δ Δ′ τ → Env ⌊ Δ_U ⌋ → .(Δ ⊆ Δ_U) → ⟦ τ ⟧
```

This *optimised semantics* shows that we can do a similar program transformation and will be useful in its correctness proof. The implementation simply maps each constructor to its counterpart in Expr, with some renaming (e.g. from $\lfloor \Delta_1 \rfloor$ to $\lfloor \Delta_1 \cup \Delta_2 \rfloor$) and the abovementioned case distinction.

```
dbe : LiveExpr Δ Δ′ τ → Expr ⌊ Δ′ ⌋ τ
dbe (Let {Δ₁} {Drop Δ₂} e₁ e₂) = injExpr₂ Δ₁ Δ₂ (dbe e₂)
...
```

As opposed to forget, which stays in the original context, here we remove unused variables, only keeping $\lfloor \Delta' \rfloor$.

Correctness We want to show that dead binding elimination preserves semantics: eval ∘ dbe ∘ analyse ≡ eval. Since we know that forget ∘ analyse ≡ id, it is sufficient to show the following:

```
eval ∘ dbe ≡ eval ∘ forget
```

The proof gets simpler if we split it up using the optimised semantics.

```
evalLive ≡ eval ∘ dbe
evalLive ≡ eval ∘ forget
```

The actual proof statements in Agda are more involved, since they quantify over the expression and environment used for evaluation. As foreshadowed in the definition of evalLive, the statements are also generalised to evaluation under any Env $\lfloor \Delta_U \rfloor$, as long as it contains the live sub-context. This gives us more flexibility when using the inductive hypothesis.

Both proofs work inductively on the expression, with most cases being a straight-forward congruence. The interesting one is again Let, where we split cases on the variable being used or not and need some auxiliary facts about evaluation, renaming and sub-contexts.

Iterating the Optimisation As discussed in section 2.1, more than one pass of dead binding elimination might be necessary to remove all unused bindings. While in our simple setting all these bindings could be identified in a single pass using strongly live variable analysis, in general it can be useful to simply iterate optimisations until a fixpoint is reached.

Consequently, we keep applying dbe ∘ analyse as long as the number of bindings decreases. Such an iteration is not structurally recursive, so Agda's termination checker needs our help. We observe that the algorithm must terminate since the number of bindings decreases with each iteration (but the last) and cannot become negative. This corresponds to the ascending chain condition in program analysis literature [15]. To convince the termination checker, we use well-founded recursion [4] on the number of bindings.

The correctness of the iterated implementation follows directly from the correctness of each individual iteration step.

3.2 λ-calculus with Let-bindings

Most functional languages are based on some variant of the λ-calculus. Extending our expression language with λ-abstractions and function application would therefore make our work more applicable to these settings and provide an additional source of bindings with new kinds of transformations.

We implemented a prototype of this extended language, and adapted dead binding elimination to accommodate the new constructors. The additional cases are very similar to the existing ones, but the possible results of evaluation now include functions. Therefore, reasoning about semantic equivalence using propositional equality requires postulating function extensionality. This does not impact the soundness of the proof and could be avoided by moving to a different setting, such as homotopy type theory [25].

While these changes were unproblematic, λ-abstractions could make other transformations more challenging, so they remain a prototype for now and are not included in our core language.

4 Timetable and Planning

4.1 Further Work

There is a large number of possible directions to explore. While working on all of them is not feasible, this section gives an overview of the most promising ones.

4.1.1 Strongly Live Variable Analysis

Instead of iterating the dead binding elimination as defined above, the same result can be achieved in a single pass using strongly live variable analysis, as explained in section 2.1.

An initial prototype still contains unresolved complications in the correctness proof. These do not seem to be fundamental limitations and could be worth investigating further.

4.1.2 Other Transformations

Inlining We will focus on the simplest form of inlining, which only replaces a variable occurrence with the corresponding declaration. Sometimes, inlining is also understood to involve removing the inlined binding or even performing beta reduction wherever inlining produces a beta redex [18], but this can be taken care of separately.

On the language we defined, inlining is always possible and preserves semantics. The analysis only needs to specify which variables should be inlined, usually based on some heuristics involving the size of declaration, number of variable occurrences and similar factors.

Let-floating It is usually advantageous for bindings to be *floated inward* as far as possible, potentially avoiding their evaluation and uncovering opportunities for other optimisations. In other cases, it can be useful to *float outward* of specific constructs [19].

To make sure that moving a binding is valid, the analysis currently only needs to ensure that scope correctness is preserved, i.e. variable occurrences are never moved above their declaration or vice versa. If we limit the transformation to moving the binding as far as possible (or alternatively just across a single constructor), bindings only need to be annotated with a single bit of information – to move or not to move. However, it might be interesting to allow more fine-grained control over the desired location, which would require a more sophisticated type for the analysis result.

Common subexpression elimination The aim of this transformation is to find subexpressions that occur multiple times and replace them with a variable that is bound to this expression and only needs to be evaluated once. For a basic implementation it suffices to try finding occurrences of expressions that are the same as the declaration of a binding already in scope. These can then be replaced by the bound variable, reducing both code size and work performed during evaluation.

Capturing the results of such a program analysis will require some changes compared to the variable liveness annotations in dead binding elimination, since being replaceable by a variable is not just a property of an expression, but also its context.

This basic version potentially misses many opportunities, since it relies on the right expressions to be available in scope. Consider for example expressions like $P + P$, where the duplicated work can only be avoided by introducing a new let-binding for P. Some additional opportunities can be exposed by a preprocessing step, breaking down expressions into sequences of small let-bindings and floating them upwards, but making this work well is tricky. One also needs to consider that these additional let-bindings affect other transformations, unless they are removed again after common subexpression elimination.

More powerful common subexpression elimination Another approach is to put more work into identifying common subexpressions and making the transformation itself introduce a shared binding at a suitable point. Making this analysis efficient is challenging, but this is not a major concern for us. Our focus is on defining the right structure for the analysis results and applying the transformation accordingly.

Local rewrites There is a number of local transformations that simply rewrite a specific pattern into an equivalent one. Most examples are always valid to be performed:

- constant folding and identities,
  e.g. $P + 0 \Rightarrow P$
- turning beta redexes into let-bindings,
  i.e. $(\lambda x.Q)P \Rightarrow \textbf{let}\ x = P\ \textbf{in}\ Q$
- floating let-bindings out of function application,
  i.e. $(\textbf{let}\ x = P\ \textbf{in}\ Q)\ R \Rightarrow \textbf{let}\ x = P\ \textbf{in}\ Q\ R$

Showing that a single application of such a rewrite preserves semantics should be straight-forward, but usually we want to do a single pass over the program that applies the rewrite wherever possible. The correctness of this operation can be shown by structural induction, but requires boilerplate code, identical for each rewrite rule. We aim to factor out the redundant parts and handle local rewrites in a uniform way.

4.1.3 Extending the Language

Adding more constructs to the language gives us access to new transformations, for example to optimise their evaluation or encode them using existing constructs. However, each of them complicates the language and comes with its own challenges. The amount of code required is generally proportional to the product of number of language constructs and number of transformations.

One example are λ-abstractions (section 3.2), which we hope to fully support later on. This would also allow us to do transformations such as removal of unused function arguments and some of the local rewrites mentioned above. We outline further potential extensions and the consequences of adding them to the language.

Recursive Bindings In a recursive let-binding, the bound variable is available in its own declaration. While this only requires a small change in the definition of the syntax tree, evaluation can now diverge. The treatment of semantics requires significant changes to account for this partiality [5] [14] [10]. Some transformations then require a form of guaranteeing termination of strict bindings, since (re)moving bindings with side effects can change the program's semantics.

Mutually recursive binding groups Since mutual recursion allows multiple bindings to refer to each other, the current approach of handling one binding at a time is not sufficient. Instead, we need to allow a list of simultaneous declarations where the scope of each is extended with a list of variables. Working with this structure is expected to be laborious. It is currently unclear whether Allais' universe of syntax can even express this construction.

On the other hand, intrinsically typed implementations of related transformations could be instructive precisely because of the complexity of the bindings involved. An example is splitting binding groups into strongly connected components.

Nonstrict bindings Languages can contain strictly evaluated, nonstrictly evaluated or both types of bindings. Once there are side effects (such as nontermination), the strictness of bindings plays an important role for semantics preservation of transformations. While inlining or floating a nonstrict let-bindings is always valid, the same is only true for strict let-bindings if they are pure, i.e. free of side effects [18].

Branching The presence of a branching construct like *if-then-else* expressions makes some analyses more interesting since control flow is not known upfront. In addition, some optimisations become more worthwile:

- pushing a binding into a single branch where it is used avoids unnecessary computation,
- bindings present in all branches can be hoisted out to reduce code size,
- having information about the possible values of expressions can allow to remove unreachable branches.

These aspects however are not the main focus of this work; most of the actual transformations are unaffected.

Datatypes with pattern matching Adding algebraic datatypes is a more ambitious change, but also provides us with a new source of bindings and related transformations. GHC for example offers a wealth of inspiration with the *case-of-case*-optimisation and others [20].

4.1.4 Restricted Language Forms

Compilers often enforce additional restrictions on the structure of their intermediate languages to simplify transformations and compilation to a machine language. Popular choices include *continuation passing style* (CPS) and *A-normal form* (ANF) [12]. GHC for example performs a transformation called *CorePrep* to ensure (among others) the invariant that all function arguments are atoms, i.e. literals or variables [23].

It could be investigated how the introduction of similar restrictions impacts the transformations we study, including their proof of correctness. Furthermore, the encoding into the restricted form itself can be studied.

4.1.5 Generalisation

Ideally, further exploration will lead to the discovery of common patterns and useful strategies for performing analysis and transformations on intrinsically typed syntax trees. One particular question is whether it is useful to always separate program analysis and transformation into individual phases using an annotated version of the syntax tree, as currently done with dead binding elimination.

4.1.6 Syntax-generic programming

To avoid boilerplate code and make our work more reusable, we are investigating the possibility to work with syntax-generic definitions of operations and proofs. Capturing common patterns and operations in the form of general helper functions could be insightful and reduce the effort of adding further transformations.

4.2 Schedule

The thesis deadline is on 09.06.2023. To allow for sufficient grading time, the thesis will be submitted until 26.05.2023, the end of week 21. Dedicating two weeks for holidays and one as buffer time, there are about 18 weeks of time available. The work can be split into four main phases, each with an estimated time frame.

4.2.1 Initial Experimentation (4 weeks)

To obtain the practical experience necessary to discover and encode general ideas, the first phase contains the following tasks:

- finish the alternate implementation of dead binding elimination with strongly live variable analysis
- implement inlining
- implement let-floating
- implement rewrite rules
- port dead binding elimination to the `generic-syntax` library by Allais et al.

At the same time, it will involve further reading to more deeply understand datatype- and syntax-generic programming, and explore new potentially relevant ideas, such as ornamentation [9] and coeffects [17].

4.2.2 Generalise and Extend (6 weeks)

Based on the practical experiences, some reflection is in order: What were pain points and common themes? Can parts of the solutions be factored out and reused, possibly using syntax-generic programming? More concretely, the envisioned tasks include:

- sketch out general ideas, both conceptually and in code
- challenge the approach by adding additional language features (primarily recursive bindings)
- implement more sophisticated transformations, e.g. common subexpression elimination

4.2.3 Optional Goals (2 weeks)

As time permits, work will continue on some of the remaining items layed out in section 4.1 that seem feasible and interesting based on the experience gained at that point. There might also be completely new ideas spawned by the previous work.

4.2.4 Writing (6 weeks)

Throughout the whole time, new ideas developed and features implemented will be documented in a draft document. This document can then be refined towards the end and serve as the basis for writing the thesis. The last week should be reserved for proofreading.

References

[1] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, jul 2018. `doi:10.1145/3236785`.

[2] Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming*, Gothenburg, 1999.

[3] Henk P. Barendregt. The lambda calculus: its syntax and semantics. In *Studies in Logic and the Foundations of Mathematics*, volume 103, 1985.

[4] Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers – an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, nov 2014. `doi:10.1017/s0960129514000115`.

[5] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), jul 2005. `doi:10.2168/lmcs-1(2:1)2005`.

[6] James Maitland Chapman. *Type checking and normalisation.* PhD thesis, University of Nottingham, 2009.

[7] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, may 2011. `doi:10.1007/s10817-011-9225-2`.

[8] Jesper Cockx. 1001 representations of syntax with binding. `https://jesper.sikanda.be/posts/1001-syntax-representations.html`, 2021. Accessed: 2022-12-16. URL: `https://jesper.sikanda.be/posts/1001-syntax-representations.html`.

[9] Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24(2-3):316–383, apr 2014. `arXiv:1201.4801`, `doi:10.1017/s0956796814000069`.

[10] Nils Anders Danielsson. Operational semantics using the partiality monad. *ACM SIGPLAN Notices*, 47(9):127–138, oct 2012. `doi:10.1145/2398856.2364546`.

[11] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. `doi:10.1016/1385-7258(72)90034-0`.

[12] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation - PLDI '93*. ACM Press, 1993. `doi:10.1145/155090.155113`.

[13] Dougal Maclaurin, Alexey Radul, and Adam Paszke. The foil: Capture-avoiding substitution with no sharp edges. October 2022. `arXiv:2210.04729`.

[14] Conor McBride. Turing-completeness totally free. In *Lecture Notes in Computer Science*, pages 257–275. Springer International Publishing, 2015. `doi:10.1007/978-3-319-19797-5_13`.

[15] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Berlin, 1999. `doi:10.1007/978-3-662-03811-6`.

[16] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 4th international workshop on Types in language design and implementation - TLDI '09*. ACM Press, 2008. `doi:10.1145/1481861.1481862`.

[17] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. *ACM SIGPLAN Notices*, 49(9):123–135, nov 2014. `doi:10.1145/2692915.2628160`.

[18] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, jul 2002. `doi:10.1017/s0956796802004331`.

[19] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. *ACM SIGPLAN Notices*, 31(6):1–12, jun 1996. `doi:10.1145/232629.232630`.

[20] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, sep 1998. `doi:10.1016/s0167-6423(97)00029-4`.

[21] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. *ACM SIGPLAN Notices*, 23(7):199–208, jul 1988. `doi:10.1145/960116.54010`.

[22] Mitchell Pickard and Graham Hutton. Calculating dependently-typed compilers (functional pearl). *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–27, aug 2021. `doi:10.1145/3473587`.

[23] André L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995. URL: `https://theses.gla.ac.uk/74568/`.

[24] Antal Spector-Zabusky, Joachim Breitner, Yao Li, and Stephanie Weirich. Embracing a mechanized formalization gap, 2019. `doi:10.48550/ARXIV.1910.11724`.

[25] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

A Ethics Quick Scan

Response Summary:

Section 1. Research projects involving human participants

P1. Does your project involve human participants? This includes for example use of observation, (online) surveys, interviews, tests, focus groups, and workshops where human participants provide information or data to inform the research. If you are only using existing data sets or publicly available data (e.g. from Twitter, Reddit) without directly recruiting participants, please answer no.

- No

Section 2. Data protection, handling, and storage

The General Data Protection Regulation imposes several obligations for the use of **personal data** (defined as any information relating to an identified or identifiable living person) or including the use of personal data in research.

D1. Are you gathering or using personal data (defined as any information relating to an identified or identifiable living person)?

- No

Section 3. Research that may cause harm

Research may cause harm to participants, researchers, the university, or society. This includes when technology has dual-use, and you investigate an innocent use, but your results could be used by others in a harmful way. If you are unsure regarding possible harm to the university or society, please discuss your concerns with the Research Support Office.

H1. Does your project give rise to a realistic risk to the national security of any country?

- No

H2. Does your project give rise to a realistic risk of aiding human rights abuses in any country?

- No

H3. Does your project (and its data) give rise to a realistic risk of damaging the University's reputation? (E.g., bad press coverage, public protest.)

- No

H4. Does your project (and in particular its data) give rise to an increased risk of attack (cyber- or otherwise) against the University? (E.g., from pressure groups.)

- No

H5. Is the data likely to contain material that is indecent, offensive, defamatory, threatening, discriminatory, or extremist?

- No

H6. Does your project give rise to a realistic risk of harm to the researchers?

- No

H7. Is there a realistic risk of any participant experiencing physical or psychological harm or discomfort?

- No

H8. Is there a realistic risk of any participant experiencing a detriment to their interests as a result of participation?

- No

H9. Is there a realistic risk of other types of negative externalities?

- No

Section 4. Conflicts of interest

C1. Is there any potential conflict of interest (e.g. between research funder and researchers or participants and researchers) that may potentially affect the research outcome or the dissemination of research findings?

- No

C2. Is there a direct hierarchical relationship between researchers and participants?

- No

Section 5. Your information.

This last section collects data about you and your project so that we can register that you completed the Ethics and Privacy Quick Scan, sent you (and your supervisor/course coordinator) a summary of what you filled out, and follow up where a fuller ethics review and/or privacy assessment is needed. For details of our legal basis for using personal data and the rights you have over your data please see the University's privacy information. Please see the guidance on the ICS Ethics and Privacy website on what happens on submission.

Z0. Which is your main department?

- Information and Computing Science

Z1. Your full name:
Matthias Heinrich Heinzel

Z2. Your email address:
[redacted]

Z3. In what context will you conduct this research?

- As a student for my master thesis, supervised by::
  Wouter Swierstra

Z5. Master programme for which you are doing the thesis

- Computing Science

Z6. Email of the course coordinator or supervisor (so that we can inform them that you filled this out and provide them with a summary):
[redacted]

Z7. Email of the moderator (as provided by the coordinator of your thesis project):
coordinator.cosc@uu.nl

Z8. Title of the research project/study for which you filled out this Quick Scan:
Analysis and Transformation of Intrinsically Typed Syntax

Z9. Summary of what you intend to investigate and how you will investigate this (200 words max):

When writing a compiler, intrinsically typed syntax trees allow to statically guarantee type- and scope-safety invariants. While this technique is relatively well-known, there has not been much focus on performing program analysis and transformations in this settings. To explore this area, I aim to:
- collect and document common program analyses and transformations for simple expression languages with binders
- develop an understanding of potentially relevant literature, e.g. datatype-generic programming on syntax trees
- implement several transformations on intrinsically typed expressions in the dependently-typed programming language Agda
- attempt machine-checked proofs of their correctness (preservation of semantics)
- explore the common patterns between the transformations and try capturing them as re-usable building blocks

Z10. In case you encountered warnings in the survey, does supervisor already have ethical approval for a research line that fully covers your project?

- Not applicable

Scoring

- Privacy: 0
- Ethics: 0
