TRANSLATING BETWEEN TYPE REPRESENTATIONS: A NEW COMPILER OPTIMIZATION TECHNIQUE

My Nguyen

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Arts

in the

Department of Computer Science

at Bryn Mawr College

Advisor: Richard A. Eisenberg
Abstract

Types are a powerful aid for computer scientists in building safe and elegant programs. However, very often the expressivity and abstraction power of certain type representations get in the way of program efficiency, both space and time wise. This paper offers a solution to that problem: separating compile-time type from run-time representations. With this technique, the compiler can translate between any two types, usually from a more complex to a simpler one to improve runtime performance, while preserving the original type on programmer’s end. The thesis provides both a formalization and a proof-of-concept implementation in the language Template Haskell with performance evaluations. The measurable success obtained from benchmarkings proves the potential of our proposed technique as a type-directed compiler optimization, with so much room for future developments. We acknowledge the incomplete proofs and possible errors in the thesis, but are confident in the ability to overcome those given additional time.
Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor, Professor Richard A. Eisenberg for providing invaluable guidance throughout this thesis, and for being a constant source of inspiration, encouragement and mentorship as I stumbled my way through college over the past 4 years. I am glad I walked into his Introduction to Computing my first semester at Bryn Mawr — it’s been a life-changing experience.

I would like to extend my thanks to all faculty members in the Department of Computer Science for the knowledge and support they have given me, especially Professor Deepak Kumar and Professor Dianna Xu. Thanks to my fellow CS majors for the time we worked and stayed up over homework together — I hope everyone the best in their future journeys. I am obliged to use this opportunity to thank my dear friends during college time — Tessa, Blossom, Lan-Ngoc, Jiwoo, Esther, Steve, Yifan, Gia-Thai; I am extremely fortunate to know you all and have your constant support. Thanks to the Bico Vietnamese Student Association for giving me a home away from home.

Lastly, I would like to thank my family, my parents in Vietnam and my family here in the US, Jeff and Jen. Thank you for your unconditional, unequivocal love and encouragement. I love you, and owe it all to you.
1 Introduction

In computer programming, types are attributes of data that enable both the users and the compiler to use data efficiently and meaningfully. For programmers, types provide an abstraction of data that makes sense for human brains, grouping data by certain representational or functional properties: for example, an integer is different from a string. This allows users not only to write programs expressively and flexibly, but also to mentally type-check them for correctness. For the compiler or interpreter, types inform them of how the data is intended to be used: how much memory is used, what is the memory layout and how to type-check. For example, in Java, if an int is declared, the compiler would know to reserve 32 bits for it, versus if the type is long, it would need 64 bits. For the programmers, they would know the value should be a whole, non-fractional number between $-2^{31}$ and $2^{31} - 1$.

So to both programmers and compilers, types serve two main purposes: dictating how the data is to be represented, interpreted and type-checked, and describing the memory layout. In most modern compilers and programming languages, these two aspects go hand-in-hand, with a few exceptions such as the two-dimensional arrays in C. To users, the 2D array looks like a matrix, but in memory, it is represented just as a 1D array with rows laid out contiguously. This discrepancy in how the data is laid out conceptually vs. actually in memory helps: the programmer on the surface can visualize and use the 2D arrays as intended, while in memory the compiler also has an easy time dealing with this linear structure. But currently, cases where the compile-time differs from runtime representations like this are few and far between, and often limited to fixed situations built into the compiler.

We therefore propose the generalization of this idea, allowing the separation of compile-time types from runtime for just about any type and any situation the users desire, as long as there is a valid translation between the runtime one and compile-time, just like between the above-mentioned 2D and 1D arrays. The translation, of course, should preserve the program’s expected results. This can be considered a compiler optimization technique, potentially reducing programs’ space allocations and running time, while preserving type expressiveness to users. The thesis introduces and formalizes such translation, provides a general framework for proving its correctness, and a proof-of-concept implementation in Template Haskell.

Expected Contributions

More specifically, the thesis expects to contribute:

- A clear definition and formalization of the translation mechanism in the environment of the simply typed lambda calculus.
- General proof framework including typing preservation and directions for proof of correctness of the technique.
- High-level discussion of an alternative type system and translation mechanism that is the basis for our implementation.
• A proof-of-concept implementation of the technique in Template Haskell and performance benchmarkings.

2 Motivation

As mentioned in the introduction, types serve to express both representational and structural properties of data and their layout in memory. As programmers’ need for expressiveness increases, however, types can grow large and complicated. For example, to simulate a single-core computer architecture in Java, we might need an object of this type:

```java
public class Simulator {
    private CPU cpu;
    private Memory mem;
    private Bus mainBus;
    ...
}
```

where the CPU class may look like:

```java
public class CPU {
    private Processor processor;
    private ControlUnit cu;
    private MemoryManagementUnit mmu;
    ...
}
public class Processor {
    private Register[] registers;
    private ArithLogicUnit alu;
    ...
}
```

Similarly, Memory class will have RAM, ROM; MemoryManagementUnit will have an array of PageTableEntries, etc., making the type Simulator highly nested. It is easy to see why this is necessary practically: the hierarchy mimics the actual structure of the computer architecture, rendering it convenient for programmers to understand, control, and debug the simulator. Yet this design incurs significant cost in the compiler: each instance’s structure is represented with layers of pointers, making it slow and costly to operate upon. For example, to simply read from a register, we will have to traverse by pointers from a simulator object to its CPU, to the CPU’s processor, to the processor’s array of registers and eventually to that one needed register. Thus here, the type’s expressiveness has slowed down the program performance unnecessarily.

But can we do this differently? The compiler actually does not need to care so much about the hierarchy; all it cares about is where the thing it needs to access lies in memory. So instead of the nested object above, the compiler would have done just fine with a flattened, linear structure that contains all lowest-level elements of the computer architecture. If we could somehow provide this translation, operation time on the structure would be significantly lowered, while on the surface the users would still be able to
benefit from the conceptual hierarchy of the type.

Types too could get complicated to satisfy theoretical expressivity, for example by providing certain mathematical properties for program verification. The Peano natural number type is one of those. It is defined recursively in Haskell as follows:

```haskell
data Nat = Zero | Succ Nat
```

where each natural number is either a zero, or the successor of another natural number. This definition is easy to apply algorithms or perform induction on. However, it is quite cumbersome to store these in memory. Take for example the natural number 5:

```haskell
Succ (Succ (Succ (Succ (Succ Zero)))
```

which is represented by a linked list of 6 elements in memory. But each natural number is an integer, and each integer in memory is represented by its binary notation. Can we do the same for these Peano natural numbers to cut down on compiling cost, while preserving their recursive structure and mathematical properties on the user’s end? In other words, if the Peano natural number type could get translated to Integer during compilation, it would reduce program’s total running time.

These examples are motivations for our work on developing and formalizing such translation that could work for any types the user wants, as long as there is a valid translation basis. For example, suppose one wants to translate from the Simulator type above to an Integer internally, as long as one can provide some reasonable mapping mathematically, the translation program will perform accordingly. Similarly, the examples above all consist of translating from a larger, more complicated type to a simpler one, but if the user somehow wants the compiler to lay out memory for a simple linear type as if it were hierarchical, they could do that with this translation function too.

3 Background

In order to efficiently understand and evaluate the proposed technique in this thesis, it is helpful to introduce relevant background issues and related literature:

3.1 Simply Typed Lambda Calculus

The simply-typed lambda calculus (STLC) is the canonical form of Typed Lambda Calculus in type theory, supporting only one type constructor \( \rightarrow \) which forms function types. We choose to work in this environment because STLC is small enough to work with within a thesis’ scope, yet provides all necessary components such as types, abstractions, etc. for the translation mechanism to work and is extensible to other more complicated systems such as System F, the underlying type system of modern programming languages such as Haskell and ML.

Grammar

STLC grammar is given in Fig. 1.
Term variables
∃x
Base types
∅ A
Terms
esi x c λx : T.t tt
Values
v c λx : T.t
Types
T A T → T
Typing contexts
∅ Γ, x : T

Figure 1: Simply-typed lambda calculus

Γ ⊢ t : T

Term typing

x : T ∈ Γ

TVAR

Γ ⊢ x : T

c : T

TConst

Γ ⊢ c : T

Γ ⊢ t1 : T1 → T2

TApp

Γ ⊢ t2 : T1

Γ ⊢ t1 t2 : T2

TABS

Γ, x1 : T1, t : T

Γ ⊢ λx1 : T1.t : T1 → T

Figure 2: Typing rules of STLC

• Types
A type T can be any base type, denoted A, or a function between any two types, formed using the type constructor →.

• Terms
A term, or expression t can be a variable x, a constant c of some type, a lambda expression, or abstraction λx : T.t1 where every occurrence of x in t1 has type T, or an application, t1 t2, where t1 is applied on t2. A value v is basically a term, but it has to be either a constant or abstraction.

• Contexts
A typing context Γ is a set containing variables and their types. A context Γ can either be empty, ∅, or is another context Γ′ extended with a new binding by the comma construction Γ′, x : T, where x /∈ dom(Γ′) and has type T.

Typing Rules
Now given a term in a context, how do we know the type of that term? A type judgement is denoted Γ ⊢ t : T, where term t is figured to have type T using information from Γ and the shape of t, as shown in Fig. 2:

• A constant c of type T will always have its type, regardless of context, hence TConst.

• A variable x, if assigned type T in Γ, then under Γ will of course have type T. This is reflected in TVAR.

• Given a term t1 and the inference Γ, x : T1 ⊢ t1 : T2, this means that given the information of x : T where x might be free variable in t1, t1 will have type T2 under
### Evaluation Rules of STLC

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 \rightarrow t_2$</td>
<td>Evaluation rules of STLC</td>
</tr>
<tr>
<td>$t_1 \rightarrow t'_1$</td>
<td>EAppT</td>
</tr>
<tr>
<td>$t_2 \rightarrow t'_2$</td>
<td>EAppV</td>
</tr>
<tr>
<td>$(\lambda x : T_1.t_2) v_2 \rightarrow {v_2/x} t_2$</td>
<td>EAbs</td>
</tr>
</tbody>
</table>

#### Figure 3: Evaluation rules of STLC

there are no particular concern to our topic, **STLC** also has a number of evaluation rules. A step of evaluation is denoted $t \rightarrow t'$, where if $t$ is the current state of a certain term at the moment, then $t'$ is the new state of that term as the machines takes a single step of computation. The rules for $t \rightarrow t'$ are as in Fig. 3.

**Evaluation Rules**

Variables are just names, and so can not be evaluated. Constants and abstractions are values, which have reached the final step of evaluation if any, and can longer “step”. The stepping rules are thus for applications, under 3 different cases:

- If some function $t_1$ evaluates to $t'_1$ in one step, then its application on $t_2$, namely $t_1 t_2$ also evaluates to $t'_1 t_2$ in one step. This is EAppT.

- If we have some term $v_1$, which can not be further evaluated, and some term $t_2$ which steps to $t'_2$, then the application $v_1 t_2$ steps to $v_1 t'_2$, as shown in EAppV.

- Now supposed we have some value $v_2$ which can not be further evaluated, and some function $\lambda x : T_1.t_2$. Applying this function on $v_2$ then will step to a substitution of all occurrences of $x$ in $t_2$ by $v_2$, denoted by $\{v_2/x\} t_2$ in rule EAbs.

Throughout our discussion of **STLC** and its extended version in this topic, we have taken liberty and assumed the system is proved to have certain properties, such as Uniqueness of types, Preservation of type under evaluation, etc. While these are critical, fundamental properties that allow any of our later developments to work, they would take a considerable amount of time and space away from the main thesis scope in order to be fully formalized. We therefore rely on, and would recommend anyone interested in this topic to consult Pierce’s *Types and Programming Languages* [1], which provides excellent explanations and proofs for **STLC**.
3.2 Observational Equivalence

The whole purpose of this compiler optimization technique, or in fact any other compiler optimizations, is to improve runtime efficiency while preserving the final outcome of the programs. Otherwise, it would mean nothing if after optimized, a program yields a result that is different from expected. Hence a vital part of our work is to ensure that the programs before translation, \( p \), and after, \( p' \), produce the same final result in all contexts.

But how do we identify that two programs will produce the same outcome regardless? Of course, we can not just judge by their syntax, since there are expressions that are different syntactically, but have the same result. For example consider two functions in Haskell:

\[
\begin{align*}
\text{addxy} & \quad : \quad \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{addxy} x y &= x + y \\
\text{addyx} & \quad : \quad \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{addyx} x y &= y + x
\end{align*}
\]

These two expressions are syntactically different, but due to commutativity of addition, actually will give the same final number in all circumstances. So instead of checking syntax, the way to make sure two programs yield the same result is simply to observe their contribution to the final outcome of any complete program in any context. This concept is called observational equivalence, the central idea to the proof of correctness of our proposed optimization technique.

Harper \([2]\) formalizes observational equivalence in his book *Practical Foundations for Programming Languages*, and we would encourage interested readers to refer to this book for formal definition and proofs around observational equivalence. For the sake of simplicity and generality in this background section, we will only focus on the core concepts of observational equivalence and assume all other formal conditions and proofs are met.

Here, we are looking at observational equivalence in *System PCF* \([2]\). It is basically an extended version of pure STLC, with one base type \text{nat} with constructor \( 0 \) and \text{Succ}, fix-point recursion, and if-conditionals with three clauses. Under this type system, Harper defines:

- A complete program to be a closed expression of type \text{nat}
- The observable behavior of a program to be the naturals it evaluates to
- An observation over an expression is the use of that expression in a complete program
- A observation context, denoted \( C \), is an expression with a hole anywhere in it acting as a place-holder for another expression, so we can make an observation about that expression.
- A program context is a closed expression context of type \text{nat}. 


• A replacement is the filling of a whole in expression context $C$ with expression $t$, denoted $C\{t\}$.

• A typing judgement $C : (\Gamma \triangleright T) \leadsto (\Gamma' \triangleright T')$ is such that if $\Gamma \vdash t : T$, then $\Gamma \vdash C\{t\} : T'$.

• A multi-step evaluation, denoted $t \red^{\star} t'$, is the single step evaluation $\red$ repeated over a number of times, where $t$ is the state prior to any step of the machine, and $t'$ is the state after.

With all core concepts defined, now we can start on the definition of observational equivalence. First, let’s look at a concept called Kleene equality:

**Definition 3.1.**
We say that two complete programs of type $\texttt{nat}$, $p$ and $p'$ are Kleene equal, written $p \simeq p'$ iff for every natural number $n \geq 0$, $p \red^{\star} n$ iff $p' \red^{\star} n$.

We can now define observational equivalence:

**Definition 3.2.**
Two expressions $t$ and $t'$ where $\Gamma \vdash t : T$ and $\Gamma \vdash t' : T$ are observationally equivalent, denoted $t \equiv t'$, iff for every program context $C : (\Gamma \triangleright T) \leadsto (\emptyset \triangleright \texttt{nat})$, $C\{t\} \simeq C\{t'\}$.

That is, two expressions of the same type are observationally equivalent if applied to any context. they produce the same naturals.

Proving observational equivalence is non-trivial because it involves co-induction, and Harper provides excellent explanations and proofs for this in his book [2] for those of interest. More discussion on observational equivalence proof in the context of our translation mechanism is provided in Section 4.3.2.

3.3 Template Haskell

*Template Haskell* [3] is a Glasgow Haskell Compiler (GHC) extension to the language *Haskell*, which supports compile-time metaprogramming. Basically, it allows users to write meta Haskell programs which are evaluated at compile-time and generates actual *Haskell* programs as a result.

We choose to implement the proof-of-concept in this language because it supports compile-time program construction and type-checking, suitable for user-directed compiler optimizations like our proposed idea.

3.4 Additional Related Works

That type expressiveness may interfere with program efficiency is not a new concern in the programming language field. Wadler [4] has noted the conflict between pattern matching and data abstraction, where pattern matching provides clarity and easy induction, but often at the cost of efficiency – while data abstraction is faster but requires type representation hiding. One example Wadler provides is between Peano natural numbers
and integers as presented in Section 2.

His solution for this conflict is a mechanism called views, which allows any data type to be viewed as one or many other free data types. While this is probably the closest related work to ours in both the problem realization and solution approach, it is different in that Wadler presented a mechanism to flexibly use and switch amongst different representations of certain data, but it must be done so case by case per user’s choice. Ours ask the user to provide an initial translation guide, but the compiler does automatic translation based on the guide in any possible part of the program. Hence, our translation is done at compile-time, as opposed to the views mechanism which does the translation at runtime.

4 Theoretical Formalization

4.1 Theoretical Mechanism & Type System

Our goal is to successfully translate terms related to one original type representation into another’s in the compiler and perform computations on the newly translated terms for improved efficiency, without changing the program’s final outcome or introducing any differences on the user's end. To this end, our proposed mechanism is to establish a function $e$, which translates from the original type to the target type, then call another function $d$ to translate the results back into their observational equivalents in the original type and return to the users. This “translating back and forth”, or “bitranslation” mechanism can be better understood visually:

![Translation scheme flow chart.](image)

That is, supposed we have a term $t$ of original type $T$ that we are going to perform the translation on. Our function $e$ will translate $t$ into a term $t'$ of the target type $T'$. Our translation is considered successful if applying $d$ on this new term $t'$ produces another term $t''$ that has the same original type $T$ and is observationally equivalent to the original term $t$. 
Updates to grammar:

Translation Set

\[ S ::= \emptyset \mid S, T \rightsquigarrow_{f, g} T' \]

Wellformedness of Translation Set

\[ \Gamma \vdash S \quad A \neq A' \]
\[ A \notin \text{dom}(S) \quad A \notin \text{codom}(S) \]
\[ A' \notin \text{dom}(S) \quad A' \notin \text{rng}(S) \]
\[ \forall c : A, g(f(c)) \equiv c \]
\[ \forall c : A', f(g(c)) \equiv c \]

Figure 5: Adding support for Type Translation

In order for this to work theoretically, we need to first establish a relationship between the original type \( T \) and target type \( T' \), which include mappings between values of type \( T \) and \( T' \)'s and vice versa so we can easily translate back and forth between the two. We keep a collection of such relationships, called the translation set \( S \). We update \( STLC \) to reflect this need in Fig. 5.

The bitranslation set, denoted \( S \), includes relationships between pairs of base types, \( A \) and \( A' \), along with mappings from values of \( A \) to \( A' \) via function \( f \), and vice versa via function \( g \). It is important to note that \( f \) and \( g \) here are not part of our extended-\( STLC \); they are not terms in the language, but functions defined outside to interact with the language just like \( e \) and \( d \). The idea is, in order to our bitranslation between some type \( A \) to \( A' \) to work correctly, we need to set up a certain number of base case mappings on constants of each type. The functions \( f \) and \( g \) are to set up such mappings, and we require that \( f \) and \( g \) are inverse up to observational equivalence for this to work. That is, given a term \( t : A \), then apply \( f \) and then \( g \) on \( t \) should give us back a term of the same type \( A \) that is observationally equivalent to \( t \).

There are a number of requirements and caveats on the addition of a new relationship into \( S \), as shown in the formation rule \( \text{SAdd} \). Some of these are conscious design choices, others are to accommodate the constraints of our chosen bitranslation mechanism:

- **Mappings are between base types:**
  Currently we only allow user-defined mappings in \( S \) to be between base types. One may think this is limiting, as it would not accommodate when we want both a mapping between the bases types \( \text{Nat} \) and \( \text{Integer} \), and another between function types \( \text{Nat} \rightarrow \text{Nat} \) to \( \text{Nat} \rightarrow \text{Int} \). However, allowing cases like this would break the compositionality of our translation functions \( e \) and \( d \), as they would not be able to know which type to translate \( \text{Nat} \rightarrow \text{Nat} \) into. So, this is disallowed in our system.

- **The working type set and the range of \( S \) must be mutually disjoint:**
  We define the working type set \( W \) of the grammar to be a collection of types whose terms are in used in the program before any translation happens. This should include certain types that could either be in the domain of \( S \), and those that are not (types that were not mapped). This set, however, must be completely disjoint from
the range of $S$. Complications arise due to our bitranslation mechanism. Consider the case where the type \texttt{Integer} is in the working set type \texttt{W} and is not in the domain of \texttt{S}, but is in the range of \texttt{S} via a mapping from \texttt{Nat}. When we see a term of type \texttt{Integer} after initial translation \texttt{e}, it is impossible to tell whether this was an original \texttt{Integer} term, or a new one generated by \texttt{e} from type \texttt{Nat}. Then whatever our \texttt{d} does with this term, it is not guaranteed to be correct, hence breaking our whole system.

But if these two sets are disjoint, when we see a term of some type \texttt{A} after \texttt{e} translation, our \texttt{d} would be able to look up whether or not this type is in the range of \texttt{S} or in the working set, and choose to translate back or not accordingly. This whole disjoint requirement is thus to ensure the correctness of \texttt{d} after \texttt{e}.

It is also critical to note on the set of types each function \texttt{e} or \texttt{d} is defined on. The domain of \texttt{e} is the working type set \texttt{W}, while the domain of \texttt{d} is $(\texttt{W} - \text{dom}(\texttt{S})) \cup \text{rng}(\texttt{S})$, since after \texttt{e} all types in domain of \texttt{S} must have become their image in the range of \texttt{S}.

This requirement, however, poses various inconveniences in actual implementation. It would require programmers to specify for each type in use, whether it belongs to the working type set or the range of \texttt{S}. On large and complicated programs with multiple modules, for example, this is cumbersome to do and hard to keep track of for programmers. Hence, while we are sticking with this requirement and this translation mechanism theoretically, we also propose a different scheme that is less restricting and closer to our actual implementation in Section 5.

- The mappings are injective:
  That is, each type in the domain uniquely maps to another type in the range of \texttt{S}. This is limiting in situations where we might want to have two types mapped to the same one. Take for example finite naturals in \texttt{Haskell}:

  ```haskell
data Fin (n :: Nat) where
  Zero :: Fin ("Succ n)
  Succ :: Fin n -> Fin ("Nat.Succ n)
```

  Just like Nat, this type too could be represented by Integer. But if we both translate Nat and Fin to Integer, how do our function \texttt{d} know which one to translate and return back to the user? This requirement, thus, is to keep \texttt{d} deterministic.

- \texttt{f} and \texttt{g} are defined on constants, or fully saturated data constructors only:
  Just for the simplication of the type system and the proofs on our bitranslation scheme, we require that \texttt{f} and \texttt{g} are to be defined on constants of the types in domain and codomain of \texttt{S}. For example for the case of Peano naturals, we will not have a straight \texttt{f} mapping between \texttt{Succ} to $(+1)$, but will have to define it on a fully applied case of \texttt{Succ}, say \texttt{Succ Zero} or any \texttt{Succ n} for example. This is understandably limiting, but allowing otherwise would mean an extension of the grammar to include casings or some form to differentiate between different constants or values or to mimic patterns, which really would pose a hard case for our bitranslation mechanism.
Again, this issue is resolved in our alternate translation mechanism and type system at Section 5, where f and g are defined differently.

- Correctness of the mappings is left to programmers to prove:
  Because users are and should be allowed to define any mapping between any types, it is also their responsibility to ensure their mappings make sense. As in the rules, a mapping is defined primarily with defining two functions f and g that operate on constants. These are user-provided and the compiler will not check the correctness of these – it will run the translation program which use f and g and if the results come out wrong because f and g were not sound, the compiler would not even know that. Correctness is ensure when f and g are inverse up to observational equivalence. That is, suppose f maps terms of type A to A', and g maps terms of type A' to A. We need to make sure for all t : A, g(f(t)) \equiv t and for all t' : A', f(g(t')) \equiv t'. It is up to the programmer to ensure the observational equivalence required here is satisfied in practice.

- The types mapped are strict:
  We limit the types in bitranslations to be strict, that is, all the base types must be strict types. This is because lazy types make it hard to formulate and prove any translation. Firstly, there is no way to ensure that either of our translation functions (back and forth) can preserve laziness. Secondly, lazy terms do not have an inductive structure, making it impossible to prove correctness as otherwise done in strict terms.

### 4.2 Translation Functions

Given language \( \lambda \) above with set of bitranslations \( S \), define translation function \( e_S : \lambda \rightarrow \lambda \) converting from the original type to the new target type in the compiler. It is defined as follows:

- On types:
  \[
  e_S(A) = A' \text{ if } A \sim_{f,g} A' \in S.
  \]
  \[
  e_S(A) = A \text{ otherwise.}
  \]
  \[
  e_S(T \rightarrow T') = e_S(T) \rightarrow e_S(T').
  \]

As noted above, domain of \( e \) is the working type set. Per its definition, \( e \) will translate any type \( A \) in the domain of \( S \) to its image, and any other \( A \) in the working type set to itself. This behavior gets recursively applied on the structure of the function type \( T \rightarrow T' \). This function \( e \) on types is deterministic, since any type base type \( A \) either gets a mapping in \( S \) or not, so \( e_S(A) \) for any \( A \) will only produce one result. The recursive case, by inductive hypothesis, comes to the base case, and thus is also deterministic.

- On terms:
  \[
  e_S(x) = x
  \]
\(e_S(c) = f(c)\) if \(T \sim_{f, g} T' \in S, \Gamma \vdash c : T\).
\(e_S(c) = c\) otherwise.
\(e_S(\lambda x : T.t) = \lambda x : e_S(T).e_S(t)\).
\(e_S(t_1 t_2) = e_S(t_1) e_S(t_2)\).

The formation of \(e\) on terms is based on \(f\). The base case here is if there is some constant \(c\) such that \(\Gamma \vdash c : T\) and \(T \sim_{f, g} T' \in S\), then \(c\) will get translated by \(f\) to its equivalents of type \(T'\). If \(\Gamma \vdash c : T\) and there doesn’t exist any \(T'\) such that \(T \sim_{f, g} T' \in S\), then \(e\) retains the original \(c\). Another interesting case is on variable, \(x\) such that \(\Gamma \vdash x : T\), where \(e\) retains the name \(x\) on the right hand side (RHS) of the equation, but change its type to \(e_S(T)\). This mechanism gets reflected in the case of lambda as well. The rest is just structural recursion.

For any \(c\), there is only one type \(T\) such that \(\Gamma \vdash c : T\) and whether \(T \in dom(S)\) is also determined, so the base case is deterministic. We also have proved above that \(e_S(T)\) is deterministic for any \(T\), so \(e\)'s behavior on all the above structures of \(t\) is also deterministic.

- **On context \(\Gamma\):**
  \(e_S(\emptyset) = \emptyset\)
  \(e_S(\Gamma, x : T) = e_S(\Gamma), x : e_S(T)\).

  Here, \(e\) applied on an empty context is just going to be empty. \(e\) applied on some context \(\Gamma, x : T\) is done recursively, resulting in \(e_S(\Gamma), x : e_S(T)\). Since for any \(T\), \(e_S(T)\) is determined, this case is also deterministic.

Similarly, \(d : \lambda \to \lambda\) is defined extendedly from \(g\) just like how \(e\) is defined based on \(f\):

- **On types:**
  \(d_S(A) = A'\) if \(A' \sim_{f, g} A \in S\).
  \(d_S(A) = A\) otherwise.
  \(d_S(T \to T') = d_S(T) \to d_S(T')\).

  As noted above, the domain of \(d\) is \((W - dom(S)) \cup rng(S)\). Per its definition, \(d\) maps any type \(A\) in range of \(S\) to its original type \(A'\) where \(A' \sim_{f, g} A \in S\), and any other type in \(W - dom(S)\) to itself. Since we require the type mappings to be injective, there is only one such \(A'\) for each such \(A\), hence the behavior here is deterministic. For any other \(A\) that is not in the range of \(S\), per our requirement for the working type set to be disjoint from the range of \(S\), simply this type \(A\) should remain itself, which gets reflected on the second equation: \(d_S(A) = A\) otherwise. The third case is just a recursion on the structure of the function type, hence also deterministic by inductive hypothesis.

- **On terms:**
  \(d_S(x) = x\)
\[
\begin{align*}
\mathbf{d}_S(c) &= \mathbf{g}(c) \text{ if } T' \sim_{f,g} T \in S, \Gamma \vdash c : T. \\
\mathbf{d}_S(c) &= c \text{ otherwise.} \\
\mathbf{d}_S(\lambda x : T.t) &= \lambda x : \mathbf{d}_S(T).\mathbf{d}_S(t). \\
\mathbf{d}_S(t_1 t_2) &= \mathbf{d}_S(t_1) \mathbf{d}_S(t_2).
\end{align*}
\]

Similar to \(e\), the most interesting case here is if with a constant \(c\) of type \(T\). Since
the mapping \(T' \sim_{f,g} T \in S\), if exists, is injective and deterministic, and based
on the assumed correctness of \(f\) and \(g\), we can call \(g(c)\) to return the original
term in type \(T'\) deterministically. All other cases follow exactly \(e\)'s, which are also
determined.

- On context \(\Gamma\):
  \[
  \begin{align*}
  \mathbf{d}_S(\emptyset) &= \emptyset \\
  \mathbf{d}_S(\Gamma, x : T) &= \mathbf{d}_S(\Gamma), x : \mathbf{d}_S(T).
  \end{align*}
  \]

This follows straight from \(e\)'s, and thus is similarly deterministic.

### 4.3 Proof of Correctness

Now that we have the definitions of \(e\) and \(d\), let’s revisit our bitranslation scheme at
Section 4.1 to see our next step in order to prove that the translation mechanism works.
Following the arrows from the initial term \(t\) in Section 4.1 we need to make sure that
\(\mathbf{d}_S(e_S(t))\) yields the same final outcome in any context, or is observationally equivalent,
to \(t\). To this end, there are two critical proofs: that the translation \(\mathbf{d}_S(e_S(t))\) preserves
the original type of \(t\), and given this typing preservation, does produce a term that gives
the same result as \(t\) in any context.

Our thesis offers a complete proof for the typing preservation, and certain guidelines
for the proof of observational equivalence for future references.

#### 4.3.1 Type Preservation

Essentially, we are interested in proving that for any term \(t\), its type is preserved after
the whole bitranslation scheme is executed. That is, for any context \(\Gamma\), \(\Gamma \vdash t : T\)
implies \(\Gamma \vdash \mathbf{d}_S(e_S(t)) : T\). Practically, this firstly is to ensure the users that their expected result
for certain computation will be returned in the expected original type, despite its having
gone through two translations inside the compiler. Secondly, this is the first step to
ensure observational equivalence, as for any two terms to produce the same outcome
in any context, they must be of the same type.

Before any actual proofwork, it is important to make sure that the functions in question
are deterministic. That is, we want to prove:

**Lemma 1.** \(e\) and \(d\) are deterministic.
Lemma 1.1. For all contexts $\Gamma$, terms $t$ and types $T \in W$:

$$\Gamma \vdash t : T$$

implies $d_S(\Gamma) \vdash d_S(t) : d_S(T)$.

Proof. We prove Lemma 1.1 by structural induction on term $t$.

- **Case**: $t = x$, i.e., $\Gamma \vdash x : T$.
  By definition of $e$ on variable, $e_S(x) = x$ where for RHS $x$, $e_S(\Gamma) \vdash x : e_S(T)$.
  So $e_S(\Gamma) \vdash e_S(t) : e_S(T)$.

- **Case**: $t = c$.
  By definition of $e$ on constants, there are 2 cases:

  - $e_S(c) = f(c)$ where $T \sim_{fg} T' \in S$.
    By well-formedness of $S$, $f(c) : T'$. By $TConst$, $e_S(\Gamma) \vdash f(c) : T'$.
    But $T \sim_{fg} T' \in S$, i.e., $e_S(T) = T'$ by definition of $e$ on types.
    So we have $e_S(\Gamma) \vdash f(c) : e_S(T)$, i.e., $e_S(\Gamma) \vdash e_S(t) : e_S(T)$.

  - $e_S(c) = c$ otherwise.
    Since $\Gamma \vdash c : T$, from $TConst$, $c : T$.
    By $TConst$, $e_S(\Gamma) \vdash c : T$.
    Since this otherwise case means there is no such type $T'$ such that $T \sim_{fg} T' \in S$ here, $e_S(T) = T$.
    So $e_S(\Gamma) \vdash e_S(t) : e_S(T)$.

In both cases of constants here, $e_S(\Gamma) \vdash e_S(t) : e_S(T)$.

- **Case**: $t = \lambda x : T_{11}. t_{12}$.
  By $TAbs$, $\Gamma \vdash t : T_{11} \rightarrow T_{12}$ where $\Gamma, x : T_{11} \vdash t_{12} : T_{12}$.
  Applying inductive hypothesis ($IH$), we have $\Gamma, x : T_{11} \vdash t_{12} : T_{12}$ implies $e_S(\Gamma, x : T_{11}) \vdash e_S(t_{12}) : e_S(T_{12})$.
  But by definition of $e$ on contexts, $e_S(\Gamma, x : T_{11}) = e_S(\Gamma), x : e_S(T_{11})$.
  So we have $e_S(\Gamma), x : e_S(T_{11}) \vdash e_S(t_{12}) : e_S(T_{12})$.
  By $TAbs$, this leads to $e_S(\Gamma) \vdash \lambda x : e_S(T_{11}). e_S(t_{12}) : e_S(T_{11}) = e_S(T_{12})$.
  But by definition of $e$ on types, $e_S(T_{11}) \rightarrow e_S(T_{12}) = e_S(T_{11} \rightarrow T_{12})$, and on functions, $e_S(\lambda x : T_{11}. t_{12}) = \lambda x : e_S(T_{11}). e_S(t_{12})$, or $e_S(t) = \lambda x : e_S(T_{11}). e_S(t_{12})$
  So $e_S(\Gamma) \vdash e_S(t) : e_S(T_{11} \rightarrow T_{12})$.

- **Case**: $t = t_{11} t_{22}$. Suppose $\Gamma \vdash t_{11} t_{22} : T$.
  By $TApp$, $\Gamma \vdash t_{11} : T_1 \rightarrow T$ and $\Gamma \vdash t_{22} : T_{11}$.
  By $IH$, $\Gamma \vdash t_{11} : T_1 \rightarrow T$ implies $e_S(\Gamma) \vdash e_S(t_{11}) : e_S(T_1 \rightarrow T)$ and $\Gamma \vdash t_{22} : T_{11}$ implies $e_S(\Gamma) \vdash e_S(t_{22}) : e_S(T_{11})$.
  But by definition of $e$ on types, $e_S(T_1 \rightarrow T) = e_S(T_1) \rightarrow e_S(T)$.
  By $TApp$, $e_S(\Gamma) \vdash e_S(t_{11}) e_S(t_{22}) : e_S(T)$.
  But by definition of $e$ on application, $e_S(t_{11} t_{22}) = e_S(t_{11}) e_S(t_{22})$, so $e_S(\Gamma) \vdash e_S(t_{11} t_{22}) : e_S(T)$.  

Proof. Determinism follows from the definitions of $e$ and $d$, explained above.  

Lemma 1.2. For all contexts $\Gamma$, terms $t$ and types $T \in W$:
$$\Gamma \vdash t : T \text{ implies } e_S(\Gamma) \vdash e_S(t) : e_S(T).$$

Proof. This proof follows similarly from Lemma 1.1's proof.

Now, we have proved that individually $e$ and $d$ on terms translates to the expected type, but really what we are interested in is a combination of both, i.e, $d_S(e_S(t))$ for any term $t$. This is the spine of our mechanism, back in Section 4.1.

Firstly, we want to prove that $d_S(e_S(T))$ works as expected on any type $T$ in the working type set $W$, that is:

Lemma 2. For all types $T \in W$, $d_S(e_S(T)) = T$.

Proof. By structural induction on $T$.

• Case: $T = A$ for some base type $A$.
  By definition of $e$ on types, we have 2 cases:
  
  - $e_S(T) = T'$ if $T \sim_{tf,g} T' \in S$.
    Then $d_S(e_S(T)) = d_S(T')$. In our requirements for the mappings, we stress on the injectivity of $S$ type mappings, as well as on the disjoint of the working type set and the range of $S$. So when $d$ sees a type $T'$ such that there exists a mapping $T \sim_{tf,g} T' \in S$, it knows to always translate $T'$ back to a unique $T$ due to injectivity, and not to itself due to the disjoint condition.
    So $d_S(e_S(T)) = T$.
  
  - $e_S(T) = T$ otherwise, i.e, there is no $T'$ such that $T \sim_{tf,g} T' \in S$.
    Then $d_S(e_S(T)) = d_S(T)$. Since $T$ is in the working type set, by our disjoint condition, it is not in the range of $S$. So there exists no type $T'$ such that $T' \sim_{tf,g} T \in S$.
    So by definition of $d$ on types, $d_S(T) = T$.

• Case: $T = T_1 \rightarrow T_2$.
  By definition of $e$ on types, $e_S(T) = e_S(T_1) \rightarrow e_S(T_2)$
  Then $d_S(e_S(T)) = d_S(e_S(T_1) \rightarrow e_S(T_2)) = d_S(e_S(T_1)) \rightarrow d_S(e_S(T_2))$ by definition of $d$ on types.
  Applying IH on $T_1$ and $T_2$, we have $d_S(e_S(T_1)) = T_1$ and $d_S(e_S(T_2)) = T_2$.
  So $d_S(e_S(T_1)) \rightarrow d_S(e_S(T_2)) = T_1 \rightarrow T_2$, which is $T$.
  So $d_S(e_S(T)) = T$.

Next, we want to make sure for typing contexts $\Gamma$, $d_S(e_S(\Gamma))$ also works as expected:

Lemma 3. For all contexts $\Gamma$, $d_S(e_S(\Gamma)) = \Gamma$.

Proof. By structural induction on $\Gamma$. 

\[\square\]
• **Case:** $\Gamma = \emptyset$.

  By definition of $e$ on context, $e_s(\Gamma) = e_s(\emptyset) = \emptyset$.

  Then $d_s(e_s(\Gamma)) = d_s(\emptyset) = \emptyset$ by definition of $d$ on context.

  So $d_s(e_s(\Gamma)) = \Gamma$.

• **Case:** $\Gamma = \Gamma', x : T$ for some context $\Gamma'$.

  By definition of $e$ on context, $e_s(\Gamma) = e_s(\Gamma', x : T) = e_s(\Gamma'), x : e_s(T)$.

  By definition of $d$ on context, $d_s(e_s(\Gamma)) = d_s(e_s(\Gamma'), x : e_s(T)) = d_s(e_s(\Gamma'), x : d_s(e_s(T))$.

  Applying $IH$ on $d_s(e_s(\Gamma'))$, we have $d_s(e_s(\Gamma')) = \Gamma'$.

  Applying Lemma 2, $d_s(e_s(T)) = T$.

  So $d_s(e_s(\Gamma'), x : d_s(e_s(T))) = \Gamma', x : T = \Gamma$.

  In other words, $d_s(e_s(\Gamma)) = \Gamma$.

\[\Box\]

Now we have all the necessary helping lemmas to prove our main concern, typing preservation. That is, we want to show that for any term $[t : T]$ where $T$ is in the working type set, after going through the whole bitranslation scheme, first through $e$ and then $d$, should come back to be a term of type $T$.

**Lemma 4.** Type preservation: For all contexts $\Gamma$, terms $t$ and types $T \in W$:

\[\Gamma \vdash t : T \text{ implies } \Gamma \vdash d_s(e_s(t)) : T\]

*Proof.*

By Lemma 1.1, we have that $\Gamma \vdash t : T$ implies $e_s(\Gamma) \vdash e_s(t) : e_s(T)$.

By Lemma 1.2, we have that $e_s(\Gamma) \vdash e_s(t) : e_s(T)$ implies $d_s(e_s(\Gamma)) \vdash d_s(e_s(t)) : d_s(e_s(T))$.

By Lemma 2, $d_s(e_s(\Gamma)) = \Gamma$, and by Lemma 3, $d_s(e_s(T)) = T$.

So $d_s(e_s(\Gamma)) \vdash d_s(e_s(t)) : d_s(e_s(T))$ means $\Gamma \vdash d_s(e_s(t)) : T$.

So $\Gamma \vdash t : T$ implies $\Gamma \vdash d_s(e_s(t)) : T$.

\[\Box\]

### 4.3.2 Observational Equivalence

Now coming back to our scheme at Section 4.1, we want to ultimately prove that for any term $t$, $d_s(e_s(t))$ returns a term, say $t''$, that yields the same outcome to $t$ in all contexts. This is so that on the programmer’s end, there will be no difference in their final result returned to them. This concept was introduced in Section 3.2, observational equivalence.

Ideally, we would like to formally define Observational Equivalence, denoted $\simeq$, for our type system, as well as prove that our programs before and after translation are observationally equivalent. However, due to time constraints as well as limitations from the chosen translating mechanism as explained in Section 4.1, we have directed our attempts to instead providing guidelines for observational equivalence proof in this case, for future references and for anyone interested in the work. So this section includes not a formal proof for observational equivalence as relevant to our scheme, but a general framework of how to get there, given sufficient time, experience and perhaps a more robust type system and mechanism.
The first and foremost thing to do, is to fully formalize Observational Equivalence in the context of our current type system. In *Practical Foundations for Programming Languages*, observational equivalence is defined on *System T* and *System PCF*, neither of which corresponds exactly to our current type system. A full re-formalization of Observational Equivalence would enclose a formal definition, as well as proofs of the equivalence’s properties on our own extended *STLC* language.

So assuming everything is full formalized, we want to prove:

\[ \forall T \in W, \forall \Gamma, \forall t : T \in \Gamma, d_S(e_S(t)) \equiv_T t. \]

One of our conditions for the mappings between types mentioned in Section 4.1 is the ensured correctness of *f* and *g*, that *f* and *g* are inverse up to observational equivalence. Because *e* and *d* are built on *f* and *g* respectively, having *g* and *f* defined correctly is the key to observational equivalence of *d_S(e_S(t)) \equiv_T t*. More specifically, if *d* acts correctly on all terms that were translated from constants in the original type, then *d_S(e_S(t)) \equiv_T t* for all term *t*. That is, we now want to prove:

\[ (\forall T' \in dom(S), \forall c : T', d_S(f(c)) \equiv_T c) \implies (\forall T \in W, \forall \Gamma, \forall t \text{ such that } \Gamma \vdash t : T, d_S(e_S(t)) \equiv_T t. \]

But as discussed in Section 3.2, observational equivalence proof is hard as it requires co-induction, which is even difficult on the simple premise *d_S(f(c)) \equiv_T c* here. Luckily there is a way out of this such that instead of having prove observational equivalence itself, we can have it proved through logical equivalence proof, which is much easier to do. This is because of the coincidence between observational equivalence and logical equivalence [2]. In this section of our thesis, we will instead try to explain logical equivalence on a higher level so we can see how it helps our proof. Following the definition of Kleene equality on type *nat* in *PCF* introduced in Definition 3.1 we define logical equivalence as follows:

**Definition 4.1.**

We say that two closed expressions of type *T*, *p* and *p'* are logically equivalent, written *p \sim_T p'* as follows:

\[
\begin{align*}
 p \sim_{nat} p' & \quad \text{iff } p \equiv_{nat} p' \\
 p \sim_{T_1 \rightarrow T_2} p' & \quad \text{iff } p_1 \sim_{T_1} p'_1 \text{ implies } p(p_1) \sim_{T_2} p'(p'_1)
\end{align*}
\]

Observably, this definition exploits type and licenses the use of single induction on type for proof. We can look at an example of logical equivalence on the simple type *nat* in *System PCF*. To prove any *E(p, p')* whenever *p \equiv_{nat} p'*, it is enough to show that:

- *E(\overline{0}, \overline{0})*, and
- if *E(p, p)*, then *E(n + 1, n + 1)*.

Hopefully, this shows how logical equivalence proof is simpler to work with than observational equivalence. Thanks to their coincidence, we can now prove observational equivalence through logical equivalence. That is, our want-to-prove now becomes:

\[ (\forall T' \in dom(S), \forall c : T', d_S(f(c)) \equiv_T c) \implies (\forall T \in W, \forall \Gamma, \forall t \text{ such that } \Gamma \vdash t : T, d_S(e_S(t)) \equiv_T t. \]

As for the premise, since we are dealing with constants of types here, logical equivalence should easily be observed through Kleene equality. Since both logical equivalence
and our translation mechanism are type-directed, the former lends itself naturally to proofwork in the latter. However, due to the time constraints and to the holes in both our translation scheme and the formalization of observational equivalence and logical equivalence for our type system, we have not produced a formal, complete proof of logical equivalence and hence observational equivalence as wanted here. Our hope for this section is that it has provided new insights into and directions for formal proof of correctness of our translation scheme for future references.

4.4 Alternative Theoretical Framework

4.4.1 Motivation

Through the research process, it has come to our attention that the original theoretical framework lacks in many aspects:

- **Feasibility:**
  Our current bitranslation scheme has a number of restrictions that render it impractical for implementation into any real world programming language. Specifically, it imposes the disjoint condition on the “working set” of types and the range of $S$, the set of types mapped. In a real world program, this would introduce a lot of complication for the users, as discussed in Section 4.1. We probably would need a system that just translates one way, and perform computation on that translated program throughout.

- **Robustness:**
  The current scheme also is very limiting in its power. For example, it imposes the mappings between the types, $f$ and $g$ to be on constants only. In real world, we should allow this to work on non-saturated constructors, such as $\text{Succ}$, or saturated constructors with variables in it such as $\text{Succ } n$, as these are important to the recursive definitions of many functions on $\text{Nat}$. Thus, we need a system that accommodates mappings between non-saturated constructors, or between patterns in functions.

This prompted us to devise a different translating scheme and type system that are more practical and robust, that support pattern and unsaturated constructor translation. Due to time constraints, we have not produced a full-fledged formalization with all rules and proofs here, and this section will be about a high level discussion of the new approach.

4.4.2 Alternative Grammar

Some highlights of this alternative scheme’s grammar is shown in Fig. 6. It is very different from our previous language’s grammar in many aspects:

- **Terms**
  Besides the basic variables, abstractions and applications, this language supports data constructors and $\text{let}$ clauses. Data constructors would allow for more flexible mappings between expressions, and $\text{let}$ binds function declarations within a scope of an expression.
**Term variables**
\[ \exists x \]

**Data constructors**
\[ \exists K \]

**Index variables**
\[ \exists i \]

**Terms**
\[ t ::= x \mid \lambda x : T.t \mid t t \mid K \mid \textbf{let} \text{DECL} \textbf{in} t \]

**Declarations**
\[ \text{DECL} ::= \overline{x_1 p_1} = t_i^i \]

**Patterns**
\[ p ::= x \mid K \overline{p} \]

**Translation set**
\[ S ::= \emptyset \mid S, T \xrightarrow{f,g} T' \]

---

**Figure 6: Alternative Type System**

- **Declarations**
  Function declarations involve a variable \( x \), which is the name of the declaration, followed by one or more equations.

- **Patterns**
  Patterns can either be a variable, or a data constructor applied to a list of patterns. The addition of patterns in this system allows it to perform pattern-match, or the way to tell between different cases in function declarations.

- **Translation set**
  The translation set seems to be the same as the STLC one, as it contains mappings between two types \( T \) and \( T' \), and two functions \( f \) and \( g \). However, the meanings of \( f \) and \( g \) are different here. Our new mechanism is one where only a single translation forward, \( e \), happens, and there is no translating back. \( f \) and \( g \) are therefore not base mappings for \( e \) and \( d \) anymore, but are mappings on expressions and patterns respectively. In some sense, there is still some forward-backward translation, as by nature of patterns, translating patterns forward is like translating expressions backward.

While we do not include a definition of \( e \) here, it will look somewhat similar to the old definition of \( e \) in its recursive structure. There are a few differences, however:

- **\( e \)** here will be dependent not just on \( S \) like before, but also on typing context \( \Gamma \). This is because of our additions of \textbf{let} bindings, where translations on \textbf{DECL} will affect the typing context of \( t \) in any clause \textbf{let} \text{DECL} \textbf{in} \( t \). Also, due to the equations in declarations, translations on patterns from the LHS of the equations will change typing contexts for the expressions on the RHS. Thus, the new signature for our function \( e \) is \( e_{S,\Gamma} \).

- **\( e \)** will be defined using both \( f \) and \( g \), where \( f \) are mappings on expressions and \( g \) are mappings on patterns. \( e \) thus will call base case \( f(t) \) for any \( t \) in the domain of \( f \), and call \( g(p) \) on any \( p \) in the domain of \( g \).

This approach, despite the lack of formalization here, actually is the basis of our implementation. It is actually devised from both our realizations of the shortcomings of the previous system theoretically, and from our attempts in an actual implementation of the technique in Template Haskell. That is why despite the lack of formalism, we still...
strongly believe this approach is the step in the direction and has much room for future developments.

5 Proof of Concept Implementation

The proof of concept is written in Template Haskell and closely follows the alternative translation scheme and type system. Specifically, the user is required to provide the set of translations, which is our set \( S \) in the alternative theory. Each element of \( S \) contains:

- Original type (\textit{from} type)
- Type after translation (\textit{to} type)
- Mappings between expressions (function \( f \) in alternative theory)
- Mappings between patterns (function \( g \) in alternative theory)

5.1 Implementation Goal

The translation program, \( E \) parallels the translating function \( e \) in our theory. Given a set of translations and any program \( q \), \( E \) will look at \( q \) and translate any occurrences of the original type, expressions and patterns into the new type, expressions and patterns. Other types and expressions will remain unchanged. Regardless of whether a change occurs, program \( q \)'s behavior should not change at all, following the proof of observational equivalence.

5.2 Scope of Implementation

5.2.1 What works

The current implementation currently is tested and works on a basic, fundamental subset of Haskell. It supports correct translation of:

- Type signatures
- Function declarations
- Expressions

with expressions of shapes:

- Variable
- Function
- Type Application
- Lambda
- Conditionals
• Tuples
• Let expression
• Infix expression
• Parenthesized expression

and patterns of shapes:
• Constructor
• As pattern
• Parenthesized pattern
• Tuple
• List of patterns
• Infix pattern
• Bang pattern
• Tilde pattern
• Pattern with signature
• View pattern

5.2.2 What doesn’t work

The program \( E \) lacks the ability to fully translate other advanced Haskell programs, such as:

• Type declarations (using \textit{data}, \textit{newtype} or \textit{type} keywords)
• Data and type family declarations
• Class and instance declarations
• Pattern synonyms
• Role annotation

It also does not take into account other Haskell subtleties such as kinds, sum arities, injectivity annotation, etc. We believe that it is doable to build a full-fledged translating scheme that works on the whole Haskell language, but it would take additional time and effort to develop and test than allowed. While the actual details will be much more complex, we believe our alternative theory and its specifications still hold valid and can provide a prototype for further expansion. For the purpose of just a proof-of-concept, we settle for the current version of \( E \) for now.
5.3 Program Demonstration

5.3.1 Non-recursive Type

Let us walk through a small yet telling example of how the translating program works as it is supposed to, following the theoretical foundations:

A Bryn Mawr student can either live off campus, or on campus in one of three dorms: Merion, Pembroke, and Denbigh. Suppose we are building a program to survey students’ living situations. We can use these data types:

```haskell
data LivingSituation = OffCampus
                      | OnCampus Dorm
  deriving (Eq, Show, Read)

data Dorm = Merion
           | Pembroke
           | Denbigh
  deriving (Eq, Show, Read)
```

Following these type declarations, a student living in dorm Merion will have her dorm situation in memory as OnCampus Merion.

Yet in reality, living in a dorm means you are on campus, and living on campus means you live in one of those dorms. Therefore, the fact that we need OnCampus Merion to represent a student living in Merion is unnecessary – just Merion alone would have implied the OnCampus part. We might as well get by with a simpler type:

```haskell
data LivingSituation' = OffCampus'
                        | Merion'
                        | Pembroke'
                        | Denbigh'
  deriving (Eq, Show, Read)
```

This would have used less memory and reduced the number of pointer traversal at runtime, compared to the first defined type. However, there might be cases where the first type might prove more useful to the users (for example: to compare the number of students living off campus vs. on campus, where we don’t need to care which specific dorms they live in), or simply, because it makes more sense intuitively that one person can either live on or off campus. Thus, we do not want to get rid of LivingSituation completely, despite its cost on performance. The translating program E can resolve this conflict by preserving LivingSituation on the user’s interface, while changing it to LivingSituation’ in the compiler to improved efficiency.

To this end, the user would have to provide the translation programs with certain instructions. For example, the users would have to declare that the from type is LivingSituation and to type is LivingSituation’, and establish certain basic expression and pattern mappings, such as, OffCampus would be translated to OffCampus’, OnCampus Merion to Merion’, etc. With this information, now our translation program can automatically translate this program that prints out the living situation information:

```haskell
showLS :: LivingSituation -> String
showLS OffCampus = "Off Campus"
showLS (OnCampus dorm) = "In the dorm " ++ show dorm
```

into an equivalent program:
Here, the variable dorm has been recognized by the translation program as having 3 possibilities: Merion, Pembroke, and Denbigh, and the newly generated program translates all these possibilities into their counterparts in LivingSituation'.

A more complicated case would be one involving more than one pattern, for example this program that verifies whether two living situations are the same:

```haskell
sameLS :: LivingSituation -> LivingSituation -> Bool
sameLS OffCampus OffCampus = True
sameLS (OnCampus d1) (OnCampus d2) = d1 == d2
sameLS _ _ = False
```

The challenge here is in recognizing that each variable d1 and d2 can be any of the three dorm options, causing that equation to have 9 possibilities if we use LivingSituation'. The translation program takes care of this also, generating an equivalent new program:

```haskell
sameLS' :: LivingSituation' -> LivingSituation' -> Bool
sameLS' OffCampus' OffCampus' = True
sameLS' Merion' Merion' = (Merion == Merion)
sameLS' Merion' Pembroke' = (Merion == Pembroke)
sameLS' Merion' Denbigh' = (Merion == Denbigh)
sameLS' Pembroke' Merion' = (Pembroke == Merion)
sameLS' Pembroke' Pembroke' = (Pembroke == Pembroke)
sameLS' Pembroke' Denbigh' = (Pembroke == Denbigh)
sameLS' Denbigh' Merion' = (Denbigh == Merion)
sameLS' Denbigh' Pembroke' = (Denbigh == Pembroke)
sameLS' Denbigh' Denbigh' = (Denbigh == Denbigh)
sameLS' _ _ = False
```

5.3.2 Recursive Type

For a demonstration on a recursive type, let's come back to our favorite Peano naturals.

```haskell
data Nat = Zero | Succ Nat
```

Let's add the most basic arithmetic operations, addition and multiplication, for our Peano naturals:

```haskell
-- addition
plus :: Nat -> Nat -> Nat
plus Zero n = n
plus (Succ m) n = Succ (plus m n)

-- multiplication
mult :: Nat -> Nat -> Nat
mult Zero n = Zero
mult (Succ m) n = plus (mult m n) n
```

We can see that each of the constructor and arithmetic operations on Peano naturals have an equivalent on integers. Zero is simply 0, and Succ is essentially incrementing by 1, the recursive plus is our simple addition, and mult our multiplication on integers. Let's bring these definitions into the scope:
We can now set up most of our translation instance between \texttt{Nat} and \texttt{Integer}:

- Original type: \texttt{Nat}
- Target type: \texttt{Integer}
- Mappings between expressions (meta-function \( f \)):
  - \( f(\text{Zero}) = 0 \)
  - \( f(\text{Succ}) = \text{inc} \)
  - \( f(\text{plus}) = \text{add} \)
  - \( f(\text{mult}) = \text{multiply} \)

This way, the recursive, non-constant functions of \texttt{plus} and \texttt{mult} will become \texttt{inc} and \texttt{multiply}, which are \( O(1) \) functions. This makes a huge asymptotic difference in runtime of any program using \texttt{plus} and \texttt{mult}.

What’s still missing here in our translation instance is mapping between patterns, or function \( g \). The nature of this mapping is more delicate and worth more explanation. Supposed we only had constants of type \texttt{Nat} as patterns, such as \texttt{Zero}, \texttt{Succ Zero}, or \texttt{Succ (Succ (Succ (Succ Zero)))}, things would be quite similar to the expression mapping: \texttt{Zero} would be \texttt{0}, and every \texttt{Succ} gets translated to an \texttt{inc} recursively on the structure. However, the gnarly point is that we also should allow patterns with variable, such as \texttt{Succ n}, or \texttt{Succ (Succ n)}, etc. The most intuitive idea is to just translate these to patterns such as \texttt{n \oplus 1} or \texttt{n \oplus 2} straightforwardly, where \texttt{\oplus} denotes infix operator of the addition \texttt{+} in pattern. However, this shape of pattern, \texttt{n \oplus k}, is forbidden in Haskell [5].

Our current implementation thus employs a hack to counter this. The idea is to translate any patterns with constructor \texttt{Succ} and variable \texttt{n} on the RHS of the equation, such as \texttt{Succ n}, or \texttt{Succ (Succ n)}, to just \texttt{n} on the RHS of the new equation, and adjust any occurrence of \texttt{n} on the LHS of the equation to its correct value according to the initial pattern. For example, if the initial pattern is \texttt{Succ n}, then any occurrence of \texttt{n} on the LHS of the newly translated equation will become \texttt{n-1}. If the initial pattern is \texttt{Succ (Succ n)}, then \texttt{n} on the LHS will become \texttt{n-2}. Implemented correctly, this hack should work efficiently and preserve correctness of the function. Currently, the hack is implemented with a recursion that counts how deep the \texttt{Succ} is applied, and accumulate that value to pass to the LHS. Another much more elegant approach would be to use \texttt{view} pattern, a feature of Haskell that allows pattern-matching against specific values,
to cleverly “expose” the \( n : + k \) nature of patterns involving constructor \( \text{Succ} \) and variable \( n \).

Now let’s look at the result of our translation schemes on expressions and function declarations. Any expression of type \( \text{Nat} \) will now be changed to \( \text{Integer} \), for example we can observe the step-by-step translation on a \( \text{Nat} \) equivalent to the number 3:

\[
\begin{align*}
(Succ (Succ (Succ Zero))) \\
(Succ (Succ (Succ 0))) \\
(Succ (Succ (inc 0))) \\
(inc (inc (inc 0))) \\
3
\end{align*}
\]

An interesting example to look at for function translation is the Fibonacci computation:

```haskell
fib :: Nat -> Nat
fib Zero = Zero
fib (Succ Zero) = Succ Zero
fib (Succ (Succ n)) = plus (fib n) (fib (Succ n))
```

This familiar example actually has a variety of expressions and patterns that can showcase our translation scheme. We have both expressions and patterns of base case \( \text{Zero} \), a constant \( \text{Succ Zero} \), a pattern involving variable \( \text{Succ (Succ n)} \), as well as application of our additive operations \( \text{Succ plus} \). Currently, the program is being translated into:

```haskell
fib' :: Integer -> Integer
fib' 0 = 0
fib' 1 = inc 0
fib' n = add (fib' (n - 2)) (fib' (inc (n - 2)))
```

We can easily verify that this corresponds exactly to a standard Fibonacci computation on integers.

5.4 Implementation Evaluations

5.4.1 Correctness

All above-mentioned programs, before and after translation, have been type-checked by the Haskell compiler as well as manually checked to make sure they are well-typed, have correct syntax and semantics, and behave no different on correct inputs.

5.4.2 Performance Benchmarkings

1. Methodology

We aim to benchmark performances, space and time-wise, on the resulting programs from the translation scheme above to prove that indeed they are optimized compared to original ones. To this end, we rely on primarily two main benchmarking tools: Criterion [6], and Glasgow Haskell Compiler (GHC)’s non-trivial runtime system [7].

Criterion is a Haskell microbenchmarking library that provides a number of measurements on program compiled with it. We are mainly concerned with two benchmarks here:
• **OLS Regression**: Using a regression model called Ordinary Least Square, this number estimates the time needed for a single execution run of the activity being benchmarked, in seconds. We need it as it measures a single execution of the program; a lower number means a faster program.

• **Mean execution time**: As its name suggests, this is a mean for the execution time, calculated by the total execution time divided by number of runs. This number usually would be really close to the OLS regression data.

GHC runtime system also measures memory storage, garbage collections, initiation time, thread scheduling, etc. We care about one benchmark:

• **Total memory allocation**: How much memory (in bytes) is allocated in the heap over the whole program run. This gives us the space measurements; lower number means more space efficiency.

It is important to note that we are measuring the space and time of programs prior and after the translation process, and not those of the translation itself. This is because our goal is to prove that after the translation, the program will run much faster with much less space. The efficiency of the translation itself is not our main concern here.

2. **Tests and Results** We performed benchmarkings on both the above demonstrations on non-recursive and recursive types.

   (a) **Non-recursive type**: Using the same example of non-recursive type above, we perform the following test: running `showLS` and `showLS'` on a list of 1 million randomized `LivingSituation` and their translated counterparts `LivingSituation'` respectively.

   Here is a summary of the result:

<table>
<thead>
<tr>
<th></th>
<th>LivingSituation</th>
<th>LivingSituation'</th>
<th>Reduction Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Mem Alloc</td>
<td>5,310 megabytes</td>
<td>4,643 megabytes</td>
<td>12.5 %</td>
</tr>
<tr>
<td>OLS regression</td>
<td>2.595 s</td>
<td>1.996 s</td>
<td>23.1 %</td>
</tr>
<tr>
<td>Mean exec time</td>
<td>2.683 s</td>
<td>1.960 s</td>
<td>27 %</td>
</tr>
</tbody>
</table>

Overall, the program after translation with `LivingSituation'` outperform the original one in all measurements. For memory, it reduces the original by about 12.5%. For time, the reduction percentage is 27%. The numbers here are noteworthy considering that we are running our examples on the very simple example between `LivingSituation` and `LivingSituation'`, which differ by a single layer of hierarchy and not in asymptotic complexity in their mappings.
(b) Recursive type: Similarly, we perform the test between original program with Peano natural numbers and its counterpart after translation with integers. We populated a randomize list of 100,000 natural numbers less than or equal to 5 in Nat and Integer, and perform an accumulative plus and add on each list to find their sum, respectively.

Here, due to the asymptotic difference in time complexities between plus and add, the result is striking:

<table>
<thead>
<tr>
<th>Total Mem Alloc</th>
<th>Nat/plus</th>
<th>Integers/add</th>
<th>Reduction Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLS regression</td>
<td>1,009,955 megabytes</td>
<td>181 megabytes</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

While we were not able to obtain the mean execution time of the original program due to be time constraints, the results on the memory and single execution time are remarkable. The original program over 100,000 Peano naturals simply is very inefficient and takes approximately 20 minutes to run, with a huge amount of memory on the heap allocated. The program after translation involves just constant folded addition over integers represented as binary numbers in the memory, and so more efficient by the percentage of 99%. This number is not out of our expectation, as the difference here lies in the linear vs. constant complexity between plus and add, and since the input size is large, such a huge difference like this is bound to happen. This marks the successful proof of our translation scheme, as a compiler optimization technique.

6 Discussion & Future Directions

6.1 Caveats

There a significant number of caveats to the work presented here. First of all is the lack of full formal definitions and/or proofs for many type systems or concepts presented here, for example: properties of STLC and of observational equivalence, formal grammar of System PCF. We touch base on a considerable number of ideas and concepts deep in the field of programming languages here, and have sacrificed depth and formalities for the sake of understandability and breadth to a general audience. We therefore have taken liberty and worked with all above-mentioned concepts as if each of them has been formalized fully in our paper, and truly this is only possible thanks to the formalism and foundations from Pierce’s Types and Programming Languages [1] and Harper’s Practical Foundations for Programming Languages [2].

The second point worth discussing is our presentations of two different theoretical frameworks for our optimization technique. This comes from two different approaches
to build our technique, both theoretically and implementationally:

- At first, we chose to conceptualize the theory part first before moving to implementation. This results in the first theoretical framework, bitranslation model with on STLC, which after some amount of time proved lacking and unsuitable for practical implementation. Still, we proceeded to prove typing preservation with it in hope this can shed light on the general scheme of translating from one type representation to another.

- After figuring out the difficulties from the first system, we set to work out the technique backwards: implementation in an actual programming language first, in this case Template Haskell, then devise theory from implementation. This results in our second theoretical framework, single-translation model with patterns and data constructors. Because the implementation was successful and this framework closely resembles actual Haskell’s type system, we believe this holds so much more potential. Yet due to time constraints, our presentation of this system is just an overview. It also does not come with any proofwork on typing preservation or observational equivalence. However, it is still worth mentioning as we believe this is the correct direction to step in for those interested.

6.2 Future Work

Considering both caveats and potential in our work presented here, there is so much more room for future advancements.

One direction would be to address the caveats above, as in, explicitly state and prove properties for all related concepts, in a suitable type system and translation mechanism. Since our first theoretical framework is proved impractical, we would suggest a pursuit instead of the second one, which has so much more potential to be developed and for our technique to be fully formalized.

Another direction is to work on the implementation of this technique into a mature programming language. Currently what we have is a proof-of-concept program in Template Haskell, but if possible, we would like to build this fully into the Glasgow Haskell Compiler, or any other compiler. From our statistical measurements of the efficiency of this compiler optimization technique, we believe an actual implementation of it in any compiler will prove useful for programmers.

7 Conclusion

Overall, the thesis has achieves what it aims to do: to provide motivation and general idea about this new compiler optimization technique of translation between different type representations, to sketch out theoretical frameworks and a proof-of-concept implementation with measurable success. It comes with a number of holes and possible errors, but this hopefully has provided insights on the process of experimenting, developing and formalizing a new technique, and leaves room for future interests. We are confident that
given sufficient amount of time and experience, the technique can be fully developed and implemented as we had hoped, bringing into the field a novel way to utilize types in compiler optimizations.
References


