Additional Material for "Unifying Data Representation Transformations"

Vlad Ureche
vlad.ureche@epfl.ch

May 23, 2014

Abstract

This report shows an end-to-end formalization of the data representation transformation mechanism in the "Unifying Data Representation Transformations" paper [16]. Since the mechanism described in the paper is targeted at the Scala programming language and the specification is written against System $F_{<:}$ with local colored type inference [11, 14] formally reasoning about the calculus is a major undertaking.

Instead, in this report we start from the simply typed lambda calculus with subtyping $\lambda_{<:}$, natural numbers and unit. We add rewriting and adapt the calculus to propagate expected type information in a mechanism inspired from local colored type inference [11]. Finally we show how the representation transformation mechanism (the convert phase) rewrites terms. We prove that, given a series of assumptions about the inject phase, type-checking a term against the updated rules produces a correct and operationally equivalent term, with a minimum number of runtime coercions introduced for the annotations given.

We finish the report by giving a series of examples which show how the code is transformed.

1 Introduction

This report will present the formal aspects of the "Unifying Data Representation Transformations" paper [16]. Before diving into details, we will briefly refresh the main ideas in the paper.

1.1 Motivation

Programs can be seen as transformations between input and output data. Inside this transformation, processing the data involves passing values from one function to another, writing the data to variables, reading it back and performing primitive operations on it, such as addition and multiplication. In this context, the question of representation naturally arises: how does the program represent the data it manipulates. The questions is particularly important when using high-level languages that isolate the programmer from the low-level processor architecture and the

realities of the platform the program is executed on. Inefficiently representing data has been shown to decrease performance and inflate the heap requirements [8, 6, 7, 4, 17]. Ideally, data should be represented in the most efficient format possible for the task. For example, when operating with integers, values should be stored in the processor registers or on the stack. Yet, certain high-level language features, such as parametric polymorphism (also called generics [3]) restrict the data representations that can be used in a low level program.

Parametric polymorphism allows programmers to abstract over the type of the data they are working with. This enables the definition of generic data structures, which operate in the same manner regardless of the data they are storing. For example, a linked list will perform the exact same lookup procedure regardless of whether the elements stored are integers or strings. This exposes a uniform interface to programmers, promoting code reuse while maintaining type safety. In the following example, written in the Scala programming language [10], the `identity` function takes any data type `T` and returns the exact same data type `T`:

```
1 def identity[T](arg: T): T = arg
2 val x: Int = identity[Int](0)
```

Yet, on the low level, different types of data use very different representations: integers are stored in general processor registers, floating point numbers use dedicated registers and references are represented as pointers to the heap memory. There is clearly a tension between the high-level uniformity and the low-level architecture specialization for each primitive type. In the simplest implementation of generics, the erasure transformation [3], data is represented uniformly both at the high level and at the low level: regardless of the type, the low level representation uses pointers to heap objects. This mandates that integers, which are best stored directly in processors registers, need to be represented as heap objects and passed by reference whenever interacting with generics. Under erasure, the previous example is translated to:

```
1 def identity(arg: Object): Object = arg
2 val x: int = unbox(indentity(box(0)))
```

Under erasure, the `box` and `unbox` data representation coercions are introduced as part of the translation. Yet, introducing coercions between data representations in an optimal fashion is a difficult task, which has been studied extensively [6, 8, 15]. Still, to this day, the algorithms for managing different data representations are custom tailored for specific tasks and do not have the means to express general representation constraints and to find the optimal points to introduce coercions such that the constraints are satisfied. The "Unifying Data Representation Transformation" paper [16] proposes a general and flexible mechanism for expressing data representation constraints in the type system and uses local type inference to automatically and optimally introduce coercions between different representations.

The general mechanism for data representation transformations uses annotated types [1, 2] to mark the representation constraints. This allows

local type inference to propagate the representation information along with the types. For example, in Scala, `Int` is considered the boxed representation while `@unboxed Int` is the unboxed integer. Explicitly propagating the representation in types allows the type system to automatically and optimally infer where coercions need to be introduced. This allows expressing a wide range of transformations in terms of the data representation transformation mechanism: (1) autoboxing, which automatically boxes and unboxes primitive types when necessary, as we have seen with generics, (2) value classes, which combine the advantages of flat structures with object-orientation, (3) speciaization, which optimizes the translation of generics and (4) multi-stage programming, which allows evaluating the result of a program in multiple cycles containg run, generate optimized code, compile and launch phases.

The data representation transformation mechanism uniformly expresses all these use cases in as a three-step transformation:

- In the injection phase, trasnformations mark the types to use alternative representations. This is done using annotatated types [1, 2];
- In the convert phase, type inference [13, 14, 11] and the annotations are used to guide optimal introduction of coercions between representations;
- Finally, having marked where and how the coercions take place, the data representation transformation can use the final semantics of the alternative data representations.

In the `identity` example, the autoboxing transformation would use the inject phase to mark values to be represented in their direct representation:

```
1 def identity[T](arg: T): T = arg
2 val x: @unboxed Int = identity[Int](0: @unboxed)
```

In the type application `identity[Int]`, the integer type is not marked as `@unboxed` since erased generics do require using boxed values. The next phase, convert, automatically introduces coercions between unboxed and boxed values, using the type system as a guide:

```
1 def identity[T](arg: T): T = arg
2 val x: @unboxed Int = unbox(identity[Int](box(0: @unboxed))
```

Finally, in the commit phase, the autiboxing transformation can apply erasure and transform the code to its final form where `int` represents an unboxed integer:

```
1 def identity(arg: Object): Object = arg
2 val x: int = unbox(identity(box(0))
```

The same mechanics can be employed by other data representation transformations as well, therefore making the mechanism an interesting object of study.

1.2 Formalization

In order to allow studying the data representation mechanism, it should have a formal specification that allows researchers to reason about it. Yet

the original paper does not include a formalization for several reasons:

- The complexity of the Scala type system and of System F with bounded quantification, System $F_{<:}$;
- The domain-specific nature of the inject and commit phase, which are significantly different from a transformation to the next;
- The constraints of the paper, namely the page limit.

In order to provide an intuitive formalization, we restrict ourselves to the simply typed lambda calculus with subtyping $\lambda_{<:}$, natural numbers and unit. To this end, we start by introducing a very simple local type inference-like mechanism to the calculus, which is done without interfering with its properties. Then we add annotated types to the language, prove progress and preservation. We then change the typing rules to account for the fact that `0`, `succ` and `pred` manipulate unboxed natural numbers. This point corresponds to the inject phase, where annotated types and their un-annotated counterparts are in a subtype relation in both directions, corresponding to a compatible state. Then, to formalize the convert phase, we remove the subtyping rules involving annotated types and introduce rewritings that add coercions. We prove by structural induction that any term that is typable in the first calculus is also typable in the second calculus, with the additional introduction of conversions. We then prove the rewritten term is operationally equivalent to the original term and prove that it produces programs that, on any given path, peform the minimum number of runtime coercions possible.

Throughout the formalization, we use the following example:

```
let identity : Nat → Nat = λarg: Nat. arg in
  let x : Nat = (identity 0) in
    unit
```

The `identity` function in the example is not generic, since the simply typed lambda calculus with subtyping $\lambda_{<:}$ does not support generics. Instead we simulate generics by not marking the type of `arg` as `@unboxed`, therefore requiring the boxing and unboxing of the argument exactly as seen in the earlier example.

The next section will present the calculus.

2 The Calculus

This section will expain the calculus necessary for the data representation transformation mechanism, and will explain all the extensions necessary to express the injection and conversion phases. The calculus does not formalize the commit phase, which depends significantly on the data representations that are converted.

2.1 Simply Typed Lambda Calculus

We start with the simply typed lambda calculus with subtyping $\lambda_{<:}$ and augment it with natural numbers in order to have primitive types which can be transformed. This is a clear step back from System $F_{<:}$ and the Scala type system, but it makes it possible to easily reason about the

t	::=		**terms** :
	|	x	*variable*
	|	$\lambda\, x$ `:` `T.` t	*abstraction*
	|	$t\ t$	*application*
	|	`0`	*constant zero*
	|	`succ` t	*successor*
	|	`pred` t	*predecessor*
	|	`unit`	*unit*
v	::=		**values** :
	|	$\lambda\, x$ `:` `T.` t	*abstraction value*
	|	nv	*numeric value*
	|	`unit`	*unit*
nv	::=		**numeric values** :
	|	`0`	*zero value*
	|	`succ` nv	*successor value*
`T`	::=		**types** :
	|	`T` $\rightarrow$ `T`	*type of functions*
	|	`Top`	*maximum type*
	|	`Nat`	*natural numbers*
	|	`Unit`	*unit type*

Figure 1: Syntax of the $\lambda_{<:}$ calculus with `Nat` and `Unit`

Desugaring:

$$t_1; t_2 \Rightarrow (\lambda x : \texttt{Unit}\ .\ t_2)\ t_1 \quad \text{where } x \notin FV(t_2)$$

$$\textit{let}\ x\texttt{:}\ \texttt{T} = e\ \texttt{in}\ t\ \Rightarrow ((\lambda x : \texttt{T}\ .\ t)\ e$$

Figure 2: Desugaring of the $\lambda_{<:}$ calculus with `Nat` and `Unit`

Evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda\ x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_1)\ v_2 \longrightarrow [x \to v_2]\ t_1 \quad \text{(E-AppAbs)}$$

$$\texttt{pred}\ \texttt{0} \longrightarrow \texttt{0} \quad \text{(E-PredZero)}$$

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ}\ t_1 \longrightarrow \texttt{succ}\ t_1'} \quad \text{(E-Succ)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred}\ t_1 \longrightarrow \texttt{pred}\ t_1'} \quad \text{(E-Pred)}$$

Figure 3: Evaluation rules of the $\lambda_{<:}$ calculus with `Nat` and `Unit`

transformation. We also augment the calculus with `unit`, to allow simple expression of the program.

One of the main missing features of this translation is parametric polymorphism, which arguably is the main motivation behind autoboxing and specialization. Yet both value classes and staging are use cases that are not motivated by generics. In fact staging is completely orthogonal to the low level implementation of generics in the language. In the motivating example, with the `identity` function, we instantiate the generic type to `Nat`:

```
let identity : Nat → Nat = λarg: Nat. arg in
  let x : Nat = (identity 0) in
    unit
```

To simulate generics in this case, the inject phase will not mark the arugment type and the return type of `identity` for unboxing, therefore forcing the boxing and respectively unboxing of the argument and return type of the function.

On the other hand, we do include subtyping and the maximum type `Top`, which allows transforming intersting examples, such as:

```
1 val x: Object = identity(0)
```

Which would be represented as (omitting the definition of `identity`):

Typing rules:

$$\frac{x : \mathtt{T} \in \Gamma}{\Gamma \vdash x \ : \ \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\ x : \mathtt{T}_1 \vdash t_2 : \mathtt{T}_2}{\Gamma \vdash (\lambda\, x\mathtt{:}\ \mathtt{T}_1\,.\ t_2) : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{T}_1 \to \mathtt{T}_2 \quad \Gamma \vdash t_2 : \mathtt{T}_1}{\Gamma \vdash t_1\ t_2 : \mathtt{T}_2} \qquad \text{(T-App)}$$

$$\Gamma \vdash 0 \ : \ \mathtt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash t_1 \ : \ \mathtt{Nat}}{\Gamma \vdash \mathtt{succ}\ t_1 \ : \ \mathtt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\Gamma \vdash t_1 \ : \ \mathtt{Nat}}{\Gamma \vdash \mathtt{pred}\ t_1 \ : \ \mathtt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{\Gamma \vdash t \ : \ \mathtt{S} \qquad \mathtt{S} \ <: \ \mathtt{T}}{\Gamma \vdash t \ : \ \mathtt{T}} \qquad \text{(T-Sub)}$$

$$\Gamma \vdash \mathtt{unit} \ : \ \mathtt{Unit} \qquad \text{(T-Unit)}$$

Figure 4: Typing rules of the $\lambda_{<:}$ calculus with `Nat` and `Unit`

Subtyping rules:

$$\mathtt{S} \ <: \ \mathtt{S} \qquad \text{(S-Refl)}$$

$$\frac{\mathtt{S} \ <: \ \mathtt{U} \quad \mathtt{U} \ <: \ \mathtt{T}}{\mathtt{S} \ <: \ \mathtt{T}} \qquad \text{(S-Trans)}$$

$$\mathtt{S} \ <: \ \mathtt{Top} \qquad \text{(S-Top)}$$

$$\frac{\mathtt{T}_1 \ <: \ \mathtt{S}_1 \quad \mathtt{S}_2 \ <: \ \mathtt{T}_2}{\mathtt{S}_1 \ \to \ \mathtt{S}_2 \ <: \ \mathtt{T}_1 \ \to \ \mathtt{T}_2} \qquad \text{(S-Arrow)}$$

Figure 5: Subtyping rules of the $\lambda_{<:}$ calculus with `Nat` and `Unit`

```
let x : Top = 0 in
  unit
```

The syntax of the calculus is given in Figure 1 and the desugaring in Figure 2. The operational semantics are given in Figure 3. Finally, the typing and subtyping rules are given in Figures 4 and 5.

For the $\lambda_{<:}$ calculus with `Nat` and `Unit` we can rely on the inversion and substitution lemmas and on the progress and preservation theorems being proven in textbooks [13]:

2.2 Expected Type Propagation and Rewriting

Having shown the simply typed lambda calculus with subtying, natural numbers and unit, we can now introduce the type propagation, which is the part we need from the local colored type inference and rewriting, which allows transforming the program during type checking. This includes adding two new elements:

- An **expected type propagation** mechanism inspired by the colored local type ingerence [11], which will later be used to propagate data representation constraints in the program. With this addition, type propagation becomes bi-directional: the syntax tree definitions have their type propagated inwards though the typing context Γ while expressions have their expected type propagated outwards by the expected type mechanism and
- A **term rewriting** extension, which is used to inject conversions at the right place, based on the divergence between the expected type and the actual type of a node. In the initial calculus in Figure 6, no term rewriting is performed: this is intentional, as we want to prove the properties of the calculus as we go. The further sections will add rewriting rules.

A natural question to ask is why not use Hindley-Milner [5, 9] full type reconstruction and use the more restricted expected type propagation technique. The key reason is that the expected type propagation mechanism allows rewriting terms as part of type checking, whereas the H-M type reconstruction only gathers constraints without updating the program. Later, if the representations do not match, this results in a set of irreconciliable constraints that prevent the type reconstruction algorithm from progressing, without clearly indicating where the constraint needs to be introduced. Another way to state this is that, using expected type propagation and rewriting, we allow the type system to rewrite program terms.

Another question that is worth answering is why not use implicit conversions which are available in the Scala type system. Indeed, implicit arguments have been fromalized in [12]. Still, this state-of-the-art calculus does not formalize the introduction of implicit conversions based on the expected type. Instead, it focuses on implicit argument resolution, which does not require type propagation. This makes the implicit calculus in [12] impossible to use directly in both our formalization and in modelling implicit conversions. We hope that our work can serve as

the basis for extending the formalization to implicit conversions, since the convert phase could be seen as a simplified and restricted mechanism for implicit conversions. Still, from an implementation point of view, the convert phase cannot use the implicit resolution mechanism in the Scala compiler, since after name resolution import statements are removed from the syntax tree, therefore making it impossible to re-create the full scope necessary for resolving implicits.

With these two extensions, the typing judgement is the following:

$$T; \Gamma \vdash t \rightsquigarrow t' : T$$

We also introduce the wildcard type $*$ in the type propagation scheme by using $*; \Gamma \vdash ....$ The wildcard type is not a valid type in the program, but is used to mark the fact that no expected type has been propagated. In this case, $*; \Gamma \vdash t \rightsquigarrow t' : T$ implies $T; \Gamma \vdash t \rightsquigarrow t' : T$. Contrarily, if a non-wildcard type is propagated, the rewritten member should conform to this expected type.

Figure 7 introduces the new typing rules. The syntax, desugaring and subtyping rules do not change. It is worth pointing out that the only way to satisfy an expected type in this version of the calculus is through the T-Sub rule, if the expression type-checks to a subtype of the expected type. This should explain why $*; \Gamma \vdash t \rightsquigarrow t' : T$ implies $T; \Gamma \vdash t \rightsquigarrow t' : T$: by applying the T-Sub rule with the reflexive S-Refl subtyping rule on the left-hand side of the implication we immediately obtain the right-hand side.

Now that we have defined the simply typed lambda calculus with subtyping, natural numbers, unit, type propagation and rewriting, we can prove progress and preservation:

We will now add annotated types to the calculus.

2.3 Annotated Types

Annotated types allow marking the values to use an alternative representation. A very general solution is to allow annotating any type:

			types :
`T`	::=		
	|	`T → T`	*type of functions*
	|	`Top`	*maximum type*
	|	`Nat`	*natural numbers*
	|	`Unit`	*unit type*
	|	`@unboxed T`	*annotated type*

This general annotation scheme is used in the Scala and Java type systems [1, 2]. Yet, the inject phase is more restrictive: it can only annotate those types that have an alternative representation. Let us assume that in our simple calculus, only natural number representation can be transfromed. In this case, there is no semantic attached to type `@unboxed (Nat → Nat)`, since there is no alternative representation for functions. Instead, we can have `@unboxed Nat → @unboxed Nat`), which means a function from unboxed integers to unboxed integers. In order to build this restriction into the type system we can define types as:

Updated typing rules:

$$\frac{x \texttt{ : } \texttt{T} \in \Gamma}{*;\Gamma \vdash x \rightsquigarrow x \texttt{ : } \texttt{T}} \quad \text{(T-Var)}$$

$$\frac{*;\Gamma,\ x \texttt{ : } \texttt{T}_1 \vdash t_2 \rightsquigarrow t_2' \texttt{ : } \texttt{T}_2}{*;\Gamma \vdash (\lambda\, x\texttt{:}\ \texttt{T}_1\texttt{ .}\ t_2) \rightsquigarrow (\lambda\, x\texttt{:}\ \texttt{T}_1\texttt{ .}\ t_2') \texttt{ : } \texttt{T}_1 \to \texttt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{*;\Gamma \vdash t_1 \rightsquigarrow t_1' \texttt{ : } \texttt{T}_1 \to \texttt{T}_2 \quad \texttt{T}_1;\Gamma \vdash t_2 \rightsquigarrow t_2' \texttt{ : } \texttt{T}_1}{*;\Gamma \vdash t_1\ t_2 \rightsquigarrow t_1'\ t_2' \texttt{ : } \texttt{T}_2} \quad \text{(T-App)}$$

$$*;\Gamma \vdash 0 \rightsquigarrow 0 \texttt{ : Nat} \quad \text{(T-Zero)}$$

$$\frac{\texttt{Nat};\Gamma \vdash t_1 \rightsquigarrow t_1' \texttt{ : Nat}}{*;\Gamma \vdash \texttt{succ}\ t_1 \rightsquigarrow \texttt{succ}\ t_1' \texttt{ : Nat}} \quad \text{(T-Succ)}$$

$$\frac{\texttt{Nat};\Gamma \vdash t_1 \rightsquigarrow t_1' \texttt{ : Nat}}{*;\Gamma \vdash \texttt{pred}\ t_1 \rightsquigarrow \texttt{pred}\ t_1' \texttt{ : Nat}} \quad \text{(T-Pred)}$$

$$\frac{*;\Gamma \vdash t \rightsquigarrow t' \texttt{ : S} \qquad \texttt{S <: T}}{\texttt{T};\Gamma \vdash t \rightsquigarrow t' \texttt{ : T}} \quad \text{(T-Sub)}$$

$$*;\Gamma \vdash \texttt{unit} \rightsquigarrow \texttt{unit} \texttt{ : Unit} \quad \text{(T-Unit)}$$

Figure 6: Typing rules of the $\lambda_{<:}$ calculus with `Nat`, `Unit`, expected type propagation and rewriting

Updated subtyping rules:

$$\texttt{S} <: \texttt{S} \qquad \text{(S-Refl)}$$

$$\frac{\texttt{S} <: \texttt{U} \quad \texttt{U} <: \texttt{T}}{\texttt{S} <: \texttt{T}} \qquad \text{(S-Trans)}$$

$\texttt{T1} <: \texttt{Top}$ only for types of `T1`, not annotated types (S-Top)

$$\frac{\texttt{T}_1 <: \texttt{S}_1 \quad \texttt{S}_2 <: \texttt{T}_2}{\texttt{S}_1 \to \texttt{S}_2 <: \texttt{T}_1 \to \texttt{T}_2} \qquad \text{(S-Arrow)}$$

$\texttt{@unboxed Nat} <: \texttt{Nat}$ (S-Nat1)

$\texttt{Nat} <: \texttt{@unboxed Nat}$ (S-Nat2)

Figure 7: Subtyping rules of the $\lambda_{<:}$ calculus with `Nat`, `Unit`, expected type propagation, rewriting and annotations

`T`	::=		**all types** :
	|	`@unboxed Nat`	*unboxed natural numbers*
	|	`T1`	*unannotated types*
`T1`	::=		**unannotated types** :
	|	$\texttt{T} \to \texttt{T}$	*type of functions*
	|	`Top`	*maximum type*
	|	`Nat`	*natural numbers*
	|	`Unit`	*unit type*

It is important to notice that, unlike the Scala type system, the simplifed set of types above does not allow annotating a type twice. While for our use case this is the natural choice, both the type systems of Java and Scala allow multiple annotations on a type.

In this calculus, it is not yet necessary to separate `Top` and annotated types. Still, we can already to this, since it is required by the data representation mechanism anyway. To do so, we give the subtyping rules in Figure 7, where `Top` is only a supertype of any `T1` but not of any `T`. The gap is filled by combining the S-Nat1 rule with S-Top: $\texttt{@unboxed Nat} <: \texttt{Nat} <: \texttt{Top}$.

The next section will present the data representation mechanism.

3 The Data Representation Mechanism

Until now, we have built the calculus necessary to express the data representation transformation. Now we can finally express the transformations

Updated typing rules:

$$*; \Gamma \vdash 0 \rightsquigarrow 0 \ : \ \texttt{@unboxed Nat} \qquad \text{(T-Zero)}$$

$$\frac{\texttt{@unboxed Nat}; \Gamma \vdash t_1 \rightsquigarrow t_1' \ : \ \texttt{@unboxed Nat}}{*; \Gamma \vdash \texttt{succ}\ t_1 \rightsquigarrow \texttt{succ}\ t_1' \ : \ \texttt{@unboxed Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\texttt{@unboxed Nat}; \Gamma \vdash t_1 \rightsquigarrow t_1' \ : \ \texttt{@unboxed Nat}}{*; \Gamma \vdash \texttt{pred}\ t_1 \rightsquigarrow \texttt{pred}\ t_1' \ : \ \texttt{@unboxed Nat}} \qquad \text{(T-Pred)}$$

Figure 8: Updated typing rules for the injection phase

themselves.

3.1 The Inject Phase

The inject phase will inject the `@unboxed` annotation where necessary, following a custom logic. During the injection phase, the `succ`, `pred` and `0` nodes are also converted to `@unboxed Nat` instead of `Nat`, to signal the fact that their results are unboxed. The updated rules for the calculus are given in Figure 8.

With these changes, we can re-state the theorems:

Looking at our earlier example:

```
let identity : Nat → Nat = λarg: Nat. arg in
  let x : Nat = (identity 0) in
    let y : Top = 0 in
      unit
```

After the inject transformation it will look like:

```
let identity : Nat → Nat = λarg: Nat. arg in
  let x : @unboxed Nat = (indentity 0) in
    let y : Top = 0 in
      unit
```

So far, the example has not changed much, a single annotation was added for `x`. The convert phase does the heavy lifting of adding explicit representation conversions.

3.2 The Convert Phase

The convert phase is the most complex phase in the transformation. It is the first phase to make full use of both the expected type propagation and the term rewriting capacity in the calculus we defined.

The most important element in the convert phase are the conversion nodes, which are introduced while rewriting the tree. Along with the conversion nodes, the values in the calculus need to be patched to account for boxed values, as shown if Figure 9. Now that explicit conversions

t	::=		**terms** :
	|	x	*variable*
	|	$\lambda\, x$ `:` `T.` t	*abstraction*
	|	$t\ t$	*application*
	|	`0`	*constant zero*
	|	`succ` t	*successor*
	|	`pred` t	*predecessor*
	|	`box` t	*box number*
	|	`unbox` t	*unbox number*
	|	`unit`	*unit*
v	::=		**values** :
	|	$\lambda\, x$ `:` `T.` t	*abstraction value*
	|	nv	*numeric value*
	|	`box` nv	*box numeric value*
	|	`unit`	*unit*
nv	::=		**numeric values** :
	|	`0`	*zero value*
	|	`succ` nv	*successor value*
`T`	::=		**all types** :
	|	`@unboxed Nat`	*unboxed natural numbers*
	|	`T1`	*unannotated types*
`T1`	::=		**unannotated types** :
	|	`T` $\rightarrow$ `T`	*type of functions*
	|	`Top`	*maximum type*
	|	`Nat`	*natural numbers*
	|	`Unit`	*unit type*

Figure 9: Full syntax of the convert phase

Evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda\, x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_1)\ v_2 \longrightarrow [x \to v_2]\ t_1 \qquad \text{(E-AppAbs)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(E-PredZero)}$$

$$\texttt{pred (succ } nv_1\texttt{)} \longrightarrow nv_1 \qquad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ } t_1 \longrightarrow \texttt{succ } t_1'} \qquad \text{(E-Succ)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred } t_1 \longrightarrow \texttt{pred } t_1'} \qquad \text{(E-Pred)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{box } t_1 \longrightarrow \texttt{box } t_1'} \qquad \text{(E-Box)}$$

$$\texttt{box (unbox } t\texttt{)} \longrightarrow t \qquad \text{(E-BoxUnbox)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{unbox } t_1 \longrightarrow \texttt{unbox } t_1'} \qquad \text{(E-UnBox)}$$

$$\texttt{unbox (box } t\texttt{)} \longrightarrow t \qquad \text{(E-UnBoxBox)}$$

Figure 10: Full evaluation rules of the convert phase

Updated subtyping rules for the convert phase:

$$\texttt{S} <: \texttt{S} \qquad \text{(S-Refl)}$$

$$\frac{\texttt{S} <: \texttt{U} \quad \texttt{U} <: \texttt{T}}{\texttt{S} <: \texttt{T}} \qquad \text{(S-Trans)}$$

$\texttt{T1} <: \texttt{Top}$ only for types of `T1`, not annotated types (S-Top)

$$\frac{\texttt{T}_1 <: \texttt{S}_1 \quad \texttt{S}_2 <: \texttt{T}_2}{\texttt{S}_1 \to \texttt{S}_2 <: \texttt{T}_1 \to \texttt{T}_2} \qquad \text{(S-Arrow)}$$

Figure 11: Subtyping rules of the convert phase

have been introduced, they must also be considered into the operational semantics, as shown in Figure 10.

To trigger the introduction of conversions into the tree, we transform representation inconsistencies into mismatching types. The first step in doing so is to eliminate the subtyping relations S-Nat1 and S-Nat2 (introduced for the inject phase in Figure 7), which ensure compatibility between representations during the inject phase. The second step is to add two new typing rules T-AdaptBox and T-AdaptUnBox that explicitly introduce the `box` and `unbox` operations, in Figure 12. Finally, the T-Box and T-UnBox operations are added to allow re-type-checking the tree after the `box` and `unbox` operations were introduced.

Following the rewriting in the convert phase, our example will become:

```
let identity : Nat → Nat = λarg: Nat. arg in
  let x : @unboxed Nat = unbox (indentity (box 0)) in
    let y : Top = box 0 in
      unit
```

In the initial example, we started from a state where `Nat` was the only numeric type. In the injection phase, we introduced the notion of `@unboxed Nat` and added `@unboxed` annotation to `x`. With this, the conversion phase added the explicit conversions necessary to have consistent data representations in the program. From a technical perspective, a later commit phase allows the compiler to give the final semantics of the `@unboxed` annotation, but since this phase is very different for each data transformation, we will not attempt to formalize it.

It must be noted that optimality is not about the minimum total number of conversions in the program. Instead, it refers to the minium number of conversions taken on any execution path (trace) through the program, modulo the constraints introduced by the inject phase. Assuming we add booleans to our calculus with the `if` expression, we can take the following example:

Updated typing rules for the convert phase:

$$\frac{x \texttt{ : } \texttt{T} \in \Gamma}{*;\Gamma \vdash x \rightsquigarrow x \texttt{ : } \texttt{T}} \quad \text{(T-Var)}$$

$$\frac{*;\Gamma,\ x \texttt{ : } \texttt{T}_1 \vdash t_2 \rightsquigarrow t_2' \texttt{ : } \texttt{T}_2}{*;\Gamma \vdash (\lambda\, x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_2) \rightsquigarrow (\lambda\, x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_2') \texttt{ : } \texttt{T}_1 \to \texttt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{*;\Gamma \vdash t_1 \rightsquigarrow t_1' \texttt{ : } \texttt{T}_1 \to \texttt{T}_2 \quad \texttt{T}_1;\Gamma \vdash t_2 \rightsquigarrow t_2' \texttt{ : } \texttt{T}_1}{*;\Gamma \vdash t_1\ t_2 \rightsquigarrow t_1'\ t_2' \texttt{ : } \texttt{T}_2} \quad \text{(T-App)}$$

$$*;\Gamma \vdash 0 \rightsquigarrow 0 \texttt{ : @unboxed Nat} \quad \text{(T-Zero)}$$

$$\frac{\texttt{@unboxed Nat};\Gamma \vdash t_1 \rightsquigarrow t_1' \texttt{ : @unboxed Nat}}{*;\Gamma \vdash \texttt{succ}\ t_1 \rightsquigarrow \texttt{succ}\ t_1' \texttt{ : @unboxed Nat}} \quad \text{(T-Succ)}$$

$$\frac{\texttt{@unboxed Nat};\Gamma \vdash t_1 \rightsquigarrow t_1' \texttt{ : @unboxed Nat}}{*;\Gamma \vdash \texttt{pred}\ t_1 \rightsquigarrow \texttt{pred}\ t_1' \texttt{ : @unboxed Nat}} \quad \text{(T-Pred)}$$

$$\frac{*;\Gamma \vdash t \rightsquigarrow t' \texttt{ : S} \qquad \texttt{S} <: \texttt{T}}{\texttt{T};\Gamma \vdash t \rightsquigarrow t' \texttt{ : T}} \quad \text{(T-Sub)}$$

$$\frac{*;\Gamma \vdash t \rightsquigarrow t' \texttt{ : @unboxed Nat}}{\texttt{Nat};\Gamma \vdash t \rightsquigarrow \texttt{box}\ t' \texttt{ : Nat}} \quad \text{(T-AdaptBox)}$$

$$\frac{*;\Gamma \vdash t \rightsquigarrow t' \texttt{ : Nat}}{\texttt{@unboxed Nat};\Gamma \vdash t \rightsquigarrow \texttt{unbox}\ t' \texttt{ : @unboxed Nat}} \quad \text{(T-AdaptUnBox)}$$

$$\frac{*;\Gamma \vdash t \rightsquigarrow t' \texttt{ : @unboxed Nat}}{*;\Gamma \vdash \texttt{box}\ t \rightsquigarrow \texttt{box}\ t' \texttt{ : Nat}} \quad \text{(T-Box)}$$

$$\frac{*;\Gamma \vdash t \rightsquigarrow t' \texttt{ : Nat}}{*;\Gamma \vdash \texttt{unbox}\ t \rightsquigarrow \texttt{unbox}\ t' \texttt{ : @unboxed Nat}} \quad \text{(T-UnBox)}$$

$$*;\Gamma \vdash \texttt{unit} \rightsquigarrow \texttt{unit} \texttt{ : Unit} \quad \text{(T-Unit)}$$

Figure 12: Typing rules of the convert phase

```
let pick : Bool → @unboxed Nat → @unboxed Nat → Top =
    λcond: Bool.
    λchoice1: @unboxed Nat.
    λchoice2: @unboxed Nat.
      if cond then
        box choice1
      else
        box choice2
    in
  unit
```

In this example, the transformation pushes the coercions as deep as possible in the tree, speculating that some of the branches will not need the coercion. Still, this is suboptimal from a global coercion count, since the translation could have been place around the if statement to have a single global coercion instead of tow. Yet, coercing each branch separately is more optimal in the following example:

```
let pick : Bool → Nat → @unboxed Nat → Top =
    λcond: Bool.
    λchoice1: Nat.
    λchoice2: @unboxed Nat.
      if cond then
        choice1 ; no coercion!
      else
        box choice2
    in
  unit
```

In the second example, the if *cond* = `true` there are no coercions in the execution trace and if *cond* = `false` there is one. One could even argue that, in case the `if` statement is duplicated, so is the coercion. This is true, but it is captured by the requirement that the annotations injected are satisfied – if the high-level annotations are suboptimal, as we have shown in the paper, so will the transformed program.

Intuitively, optimality comes from the fact that the convert phase will propagate expected types and will not filp the representation without a constraint. Since constraints are produced by interacting with code annotated in the inject phase, we can state that conversions are optimal modulo the constraints introduced by the inject phase.

The reminder of this document has hand-written proofs attached.

Proofs for the "Unifying Data Representation Transformations" Paper

Vlad Ureche

The following proofs describe both the inject (§3.1) and the convert phase (§3.2). Any difference is noted both in the theorem statement and in the theorem proof.

Inversion lemma

① If $*;\Gamma \vdash r \rightsquigarrow x : R$ then $r = x$ and $x : R \in \Gamma$

② If $*;\Gamma \vdash r \rightsquigarrow \lambda x : T_1 . t_2' : R$ then $\exists T_2, t_2$ s.t. $R = T_1 \to T_2$
and $r = \lambda x : T_1 . t_2$
and $*; x : T_1, \Gamma \vdash t_2 \rightsquigarrow t_2' : T_2$

③ If $*;\Gamma \vdash r \rightsquigarrow t_1' t_2' : R$ then $\exists T_1, t_1, t_2$ s.t. $r = t_1 t_2$
and $*;\Gamma \vdash t_1 \rightsquigarrow t_1' : T_1 \to R$
Careful! → and $T_1, \Gamma \vdash t_2 \rightsquigarrow t_2' : T_1$

④ If $*;\Gamma \vdash r \rightsquigarrow 0 : R$ then $r = 0$ and $R = @\text{unboxed Nat}$

⑤ If $*;\Gamma \vdash r \rightsquigarrow \text{succ } t_1' : R$ then $\exists t_1$ s.t. $r = \text{succ } t_1$
and $R = @\text{unboxed Nat}$
and $@u\ \text{Nat};\Gamma \vdash t_1 \rightsquigarrow t_1' : @u\ \text{Nat}$

⑥ If $*;\Gamma \vdash r \rightsquigarrow \text{pred } t_1' : R$ then $\exists t_1$ s.t. $r = \text{pred } t_1$
and $R = @\text{unboxed Nat}$
and $@u\ \text{Nat};\Gamma \vdash t_1 \rightsquigarrow t_1' : @u\ \text{Nat}$

⑦ If $*;\Gamma \vdash r \rightsquigarrow \text{unit} : R$ then $r = \text{unit}$
and $R = \text{Unit}$

⑧ If $T;\Gamma \vdash r \rightsquigarrow t' : R$ then $\exists S.\ s.t.\ S <: R$ s.t. ②
$*;\Gamma \vdash r \rightsquigarrow t' : S$.

for the coerce phase:

⑨ If $*;\Gamma \vdash r \rightsquigarrow \mathrm{box}\, t' : R$ then $\exists t$ s.t. $R = \mathrm{Nat}$
and $r = \mathrm{box}\, t$
and $*;\Gamma \vdash t \rightsquigarrow t' : @\,\mathrm{unboxed\ Nat}$

⑩ If $*;\Gamma \vdash r \rightsquigarrow \mathrm{unbox}\, t' : R$ then $\exists t$ s.t. $R = @\,\mathrm{unboxed\ Nat}$
and $r = \mathrm{unbox}\, t$
and $*;\Gamma \vdash t \rightsquigarrow t' : \mathrm{Nat}$

⑪ If $T;\Gamma \vdash r \rightsquigarrow \mathrm{box}\, t' : R$ then $\exists t$ s.t. $R = \mathrm{Nat}$
and $T = \mathrm{Nat}$
and $r = t$
and $*;\Gamma \vdash t \rightsquigarrow t' : @\,\mathrm{unboxed\ Nat}$

⑫ If $T;\Gamma \vdash r \rightsquigarrow \mathrm{unbox}\, t' : R$ then $\exists t$ s.t. $R = @\,\mathrm{unboxed\ Nat}$
and $T = @\,\mathrm{unboxed\ Nat}$
and $r = t$
and $*;\Gamma \vdash t \rightsquigarrow t' : \mathrm{Nat}$

Theorem 1 [Uniqueness of Types And Rewritings]

Each term t has at most one type (and one rewriting). That is, if t is typeable, then its type is unique. For injection only: Moreover, there is only one derivation of this typing built from the inference rules. For convert: We can have multiple typing derivations, but we will prove a weaker property, namely the idempotence property.

A mathematical specification is:
if $Z;\Gamma \vdash t \rightsquigarrow t' : T$ and $Z;\Gamma \vdash t \rightsquigarrow t'' : T'$ [OR]
if $*;\Gamma \vdash t \rightsquigarrow t' : T$ and $*;\Gamma \vdash t \rightsquigarrow t'' : T'$ [then] [$T = T'$ and $t' = t''$]

Proof Structural induction based on the relation that:
$*;\Gamma \vdash t \rightsquigarrow t' : T$

For inject: Lemma: The rewrites produce the same terms $\Rightarrow t' =$
For convert: uniqueness of type and rewriting

① $*;\Gamma \vdash t \rightsquigarrow x : T \overset{\text{inv.}}{\Rightarrow} t = x$ and $\underline{x:T \in \Gamma}$ ③

if $*;\Gamma \vdash x \rightsquigarrow t'' : T' \overset{\text{T-Var}}{\Rightarrow} t'' = x$ and $T' = T$

② $*;\Gamma \vdash t \rightsquigarrow \lambda x:T_1.t_2' : T \overset{\text{inv}}{\Rightarrow} \exists T_2, t_2$ s.t. $T = T_1 \to T_2$
and $t = \lambda x:T_1.t_2$
and $*;x:T_1,\Gamma \vdash t_2 \rightsquigarrow t_2' : T_2$ (1)
iH. $\Rightarrow$ T_2 and t_2 are unique

if $*;\Gamma \vdash \lambda x:T_1.t_2 \rightsquigarrow t'' : T'$ + (1) $\overset{\text{T-Abs}}{\Rightarrow}$
$t'' = \lambda x:T_1.t_2'$
$T' = T_1 \to T_2$

③ $*;\Gamma \vdash t \rightsquigarrow t_1' t_2' : T \overset{\text{inv}}{\Rightarrow} \exists T_1, t_1, t_2$ s.t. $t = t_1 t_2$
and $*;\Gamma \vdash t_1 \rightsquigarrow t_1' : T_1 \to T$ (2)
careful! $\longrightarrow$ and $T_1;\Gamma \vdash t_2 \rightsquigarrow t_2' : T_1$ (3)
by the induction hypothesis (IH) $\Rightarrow$
t_1' and t_2' are unique. (4)
$T_1 \to T$ and T_1 are unique (5)

if $*;\Gamma \vdash t_1 t_2 \rightsquigarrow t'' : T' \overset{\text{T-App}}{\underset{2\text{-}5}{\Rightarrow}} t'' = t_1' t_2'$ and $T'' = T$

④ $*;\Gamma \vdash t \rightsquigarrow 0 : T \overset{\text{inv}}{\Rightarrow} t = 0$ and $T = @\,\text{unboxed Nat}$

if $*;\Gamma \vdash 0 \rightsquigarrow t'' : T'' \overset{\text{T-Zero}}{\Rightarrow} t'' = 0$ and $T' = T = @\,\text{unboxed Nat}$

⑤ $*;\Gamma \vdash t \rightsquigarrow \text{succ}\, t_1' : T \overset{\text{inv}}{\Rightarrow} \exists t_1$ s.t. $t = \text{succ}\, t_1$
and $T = @\,\text{unboxed Nat}$
and $@\,\text{unboxed Nat};\Gamma \vdash t_1 \rightsquigarrow t_1' : @\,\text{unboxed Na}$
by the iH $\Rightarrow$ t_1' is unique, can't type to any other type. (6)

if $*;\Gamma \vdash \text{succ}\, t_1 \rightsquigarrow t' : T' \overset{\text{T-Succ}}{\underset{6}{\Rightarrow}} t' = \text{succ}\, t_1'$
$T' = T = @\,\text{unboxed Nat}$

⑥ pred → identical to Succ

⑦ unit → identical to 0

(8) $\mathcal{Z}; \Gamma \vdash t \rightsquigarrow t' : T$ (4)

(also covers cases (11) and (12) of the inversion lemma)

im this case, we actually have 3 sub-cases:

(8.1) $\exists S$ s.t. $S <: \mathcal{Z}$ s.t.

$*; \Gamma \vdash t \rightsquigarrow t' : S$ and $*; \Gamma \vdash t \rightsquigarrow t'' : S'$ $\Rightarrow^{IH}$ $t' = t''$ and $S = S'$ (7)

and $T = \mathcal{Z}$.

if $\mathcal{Z}; \Gamma \vdash t \rightsquigarrow t''' : T'$

Only for convert, since im inject the only rule for expected types is T-Sub.

Q: Can another case apply?

A: No. if we try to apply T-AdaptBox, it means $\mathcal{Z} = Nat$ and $S = @unboxed\ Nat$, but $S <: \mathcal{Z}$ is not satisfied, since im convert we eliminated S-Nat1 and S-Nat2. Same reasoning for T-AdaptUnBox.

$\Rightarrow^{T\text{-}Sub}_{I}$ $t''' = t'$ and $T = T'$

(8.2) only for convert: $t' = box\ t_0'$, by the inversion lemma $\Rightarrow T = \mathcal{Z} = Nat$ and $*; \Gamma \vdash t \rightsquigarrow t_0' : @unboxed\ Nat$ (8)

if $Nat; \Gamma \vdash t \rightsquigarrow t'' : T'' \Rightarrow^{T\text{-}AdaptBox}_{8 \mathcal{Z} *}$

$t'' = box\ t_0' = t'$

$T'' = Nat = T$

* Q: is T-Sub applicable?

A: No, since $@unboxed\ Nat \not<: Nat$

(8.3) only for convert: similar to (8.2)

(9) if $*; \Gamma \vdash t \rightsquigarrow box\ t_0' : T \Rightarrow^{inv} \exists t_0$ s.t. $t = box\ t_0$ and $T = No$

and $*; \Gamma \vdash t_0 \rightsquigarrow t_0' : @unboxed\ Nat$, $*; \Gamma \vdash t_0 \rightsquigarrow t_0'' : T''$ $\Rightarrow^{IH}$ $t_0'' = t_0'$ and $T'' = @unboxed\ Nat$ (9)

$*; \Gamma \vdash box\ t_0 \rightsquigarrow t'' : T' \Rightarrow^{T\text{-}Box}_{(9)}$ $t'' = box\ t_0' = t'$ and $T' = T = Nat$

(10) if $*; \Gamma \vdash t \rightsquigarrow unbox\ t_0' : T \Rightarrow \ldots$ exactly the same as (9)

$\Rightarrow$ we have covered all cases ∎

4 Conclusion

This report has shown the formalization of the "Unifying Data Representation Transformations" paper and aims at providing a base for other formalizations as well, especially in the area of implicit conversions (views).

References

[1] JSR 308: Annotations on Java Types.

[2] SIP-5 - Internals of Scala Annotations.

[3] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *OOPSLA* (1998), ACM.

[4] Dragos, I., and Odersky, M. Compiling Generics Through User-Directed Type Specialization. In *ICOOOLPS* (Genova, Italy, 2009).

[5] Hindley, R. The Principal Type-scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society* (1969).

[6] Jones, S. L. P., and Launchbury, J. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In *Functional Programming Languages and Computer Architecture* (1991), Springer.

[7] Kennedy, A., and Syme, D. Design and Implementation of Generics for the .NET Common Language Runtime. In *PLDI* (2001).

[8] Leroy, X. Unboxed Objects and Polymorphic Typing. In *PoPL* (1992), ACM.

[9] Milner, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* (1978).

[10] Odersky, M., Spoon, L., and Venners, B. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 2008.

[11] Odersky, M., Zenger, M., and Zenger, C. Colored Local Type Inference. In *PoPL* (2001), ACM.

[12] Oliveira, B. C., Schrijvers, T., Choi, W., Lee, W., and Yi, K. The Implicit Calculus: A New Foundation for Generic Programming. In *PLDI* (2012), ACM.

[13] Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.

[14] Pierce, B. C., and Turner, D. N. Local Type Inference. *ACM TOPLAS* (2000).

[15] Thiemann, P. J. Unboxed Values and Polymorphic Typing Revisited. In *Functional Programming Languages and Computer Architecture* (1995), ACM.

[16] Ureche, V., Burmako, E., and Odersky, M. Unifying Data Representation Transformations. Tech. rep., EPFL, 2014.

[17] Ureche, V., Talau, C., and Odersky, M. Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations. In *OOPSLA* (2013).