INTRODUCING TYPE PROPERTIES

Per-Lorean Graph

August 4, 2021

ABSTRACT

1 Introduction

Programming languages provide entities called language constructions: statements, expressions, and so forth; the very entities programmers use to write programs. In type theory, one thing we do is we reason about programs, so about these entities. Strictly speaking, we are equipped with meta-theoretical judgments that express certain statements about language constructions; as an example, a typical judgment found in any modern type theory is type typing judgment, ofter written as $\Gamma \vdash e : T$, which states that the expression e, under the typing context Γ, is of the type T.

To encounter more broad reasoning about these constructions, i.e., to express more ideas and assert stronger statements, we need to provide more information about them. In type theory, we usually do this by merely annotating constructions we reason about with additional information and manipulating this information in our inference rules to signify its meaning. This can be illustrated by some recently-proposed substructural type systems that, to indicate how many times a variable is used computationally, annotate one, in a typing context, with its usage information — an element of a semiring — 0, 1, or σ:

$$x_1 \overset{0}{:} S_1,\ x_2 \overset{1}{:} S_2,\ \ldots \vdash M \overset{\sigma}{:} T$$

The ability to explicitly impose such restrictions empowers us to reason about variables as resources since the only difference between ordinary data and a resource is that the latter is subject to further usage constraints. Thus, by simply attaching additional information to our language constructions, we augment the theory with a beneficial notion of resource-awareness.

Another practical notion usually introduced in a similar manner is the notion of a computational effect; effect systems, to add one, attach some additional data, F, to their typing judgments: $\Gamma \vdash e : T, F$, to express the idea that the expression e has the effect F on the environment when computed.

Further, to slightly generalize this and cover a considerable number of modal typing disciplines at once, some generalized modal type systems have been proposed. Their striking feature is modal types, obtained by simply annotating types with an additional, generic, partially ordered monoid. As a consequence of that, they greatly enlarge the expanse of practical features augmented by this pattern.

Therefore, providing more information about our language constructions in this way can be seen as a unified way to endow type theories with new useful features. In the rest of this paper, we will refer to data annotated to language constructions in this way as to static annotations.

The problem is, when willing to endow a theory with a feature following this pattern, most often, the whole theory becomes subject to a complete alteration. To encounter linear types, for instance, most theories, with their inference rules, need to be completely refined. Certainly, at the first time, it can work. However, provided that the theory stands as a foundation of a practical programming language, uncertainties in the idea of refining the whole theory whenever we need to encounter a new feature, provided that we follow a descriptive pattern to add one, arise; a much better approach would be to somehow allow a programmer on its own annotate its expressions with additional data and to decide how to interpret the data thereof.

The latter is not a huge problem, however. In programming, to signify one's meaning, we usually express it in terms of existing language constructions; that is, we write code that interprets it. This corresponds to how we, in type theory, manipulate supplement information in inference rules to designate its meaning, with the exception that the expanse of possible interpretation of the data is limited to these existing language constructions. Thus, given such a solution, and provided that existing language constructions are capable of interpreting those 0, 1, and σ as usage constraints [1], to encounter the same linear (and even quantitative) types, there is no more need to refine the whole theory; instead, we could express them as functionality in a separate program module and include one whenever a need arises.

In this paper, we present such a solution; we present a new programming language concept that we call *type properties*. The striking feature of this concept is that, under the hood, it uses types to handle these static annotations. Another discernible attribute is that we have attempted to make the work of static annotations as to their corresponding type-theoretic one; that is, when a user annotates some additional data using our concept, they are handled in a way as these annotations would be merely built-in into the type theory of a language. This means that, for instance, as type systems usually describe the static part of a language, all these static annotations must be given meaning already at the stage of compilation. This is crucial, for instance, to check the same linearity laws during compile-time.

We summarize our **contributions** in the following list:

- We provide an informal description of the concept and develop the intuition about our approach (Section 2), thus preparing readers for later sections whose content is merely technical. While describing, we define so-called *propertied types*, the very types we use to keep, retrieve, and propagate static annotations.
- We present a simple type theory, namely, $\lambda_{\twoheadrightarrow_p}$, augmented with the notion of type properties, thus formalizing our concept (Section 3).
- We then prove some important properties of $\lambda_{\twoheadrightarrow_p}$ (Section 4). In particular, we prove that the transformation, a phase where we handle all propertied types, is deterministic, that all propertied types are handled before runtime, and that the type system is type sound.
- To our knowledge, this is the first concept that takes care of expressing static annotations individually and stands as a solution to generalize the mentioned type-theoretic pattern to provide more information about language constructions. Nevertheless, there have been some approaches where these static annotations are expressible. We discuss in which aspects they are different from us and explore what is novel about our concept (Related Work).

2 Type Properties

Presenting our solution as a programming language concept means equipping a language with novel constructions that reflect our solution's ideas. In our case, this means we are obliged to provide new constructions for: annotating another construction with some additional data, retrieving these data by our need, and specifying the data's meaning.

To put it in another way, here is what we want. Firstly, we want our compiler to internally remember that we associate some additional data with an expression. This corresponds to annotating expressions we reason about with some additional information in type theory. The second wish is in the ability to be aware of whether an expression has some associated data with it at any point of the program and, consequently, to retrieve such data. Finally, since, in programming, to designate one's meaning, we tend to express it in terms of existing language constructions, i.e., we write code that interprets it, we wish to be able to write code that will tell a compiler what shall it do when it sees an expression that is annotated with such data.

Moreover, as we already mentioned in the introduction, we address this problem primarily with the type-theoretic perspective. This means that when there are some data annotated by means of our concept, for a user, it should look as if the data's presence would be built-in into the language's type theory. From this, we can infer two things. Firstly, in type theory, to provide more information about a construction, we typically associate it with the construction in our contexts, thereby making the information's presence implicit for programmers. A case in point is the same substructural type systems:

[1] Which is, in fact, not that difficult to achieve. A possible solution would be to add the possibility of "overloading" a variable's usage behavior (variable as in software engineering, not as in type theory). That is, the ability to write a function invoked when a variable is used, which allows us to keep track of one's usage. Provided that such language is equipped with ordinary destructors from the RAII world and means to stop the compilation, we could compare the user's number with the quantity we initially annotated to the variable (those 0, 1, and σ). If they mismatch in the destructor (or even in the function where we keep track of its usage), we terminate the compilation.

$$x \overset{!}{:} \text{Int},\ y \overset{!}{:} \text{Int} \vdash x + y \overset{\sigma}{:} T$$

Therefore, a programmer, in case it is not the one who explicitly imposes usage restrictions, when writes the expression $x + y$, is not aware that we hiddenly collect the information about variables' usage. The same must hold for our constructions; we must design them that for a user, it shall look that a compiler magically remembers what data we associate with which expressions. Secondly, since type systems usually define the static part of programming languages, the additional data must be given meaning already at the stage of compilation. In other words, we should evaluate all these constructions during compile-time.

This section serves two purposes. The first is to attempt to describe what these constructions might be. The second is to build a proper intuition on our approach; we will smoothly skim through their obligations and informal descriptions to proper type theoretical notations, thereby preparing the reader for Section 3, whose content, consisting only of inference rules and their descriptions, is merely technical.

2.1 Constructions

The best way to create the proper impression about programming language constructions is to illustrate them in examples. Consider the following pseudo-code snippet:

```
1  let x = 1 in
2  let x' = set(x, 1) in
3  x + x' + get(x')
```

Following our intuition, we can think of the expression $set(x, 1)$ as an appeal to the compiler to treat this expression as just x, but also remember that we associate with it the additional data of the integer 1. Thereafter, the construction $get(x')$, on the third line, is an appeal to substitute it with the data we previously requested to remember about x'. Thus, we expect this program, right before we run it, to look like the following:

```
1  let x = 1 in
2  let x' = x in
3  x + x' + 1
```

Seem to be effortless and, to be honest, useless: almost any modern programming language provides facilities to define and substitute constants at the stage of compilation. Thus, for demonstrational purposes, let us investigate a more suggestive example. For instance, consider that, in our program, we agree that if there is an integer constant associated with an expression of the integer type, then it means that this expression, at runtime, evaluates to the value of this constant. For this to have any sense, we must be able to be acknowledged that an expression has some associated data at any point of the program, to later instruct the compiler with what to do when it finds out that there is some data associated. For instance, we must be able to retrieve this information even in a function's body, which we intend to execute at runtime:

```
1  func greater_than_five (x : Int) with
2    if-has x n : Int then n > 5
3    else x > 5 in
4
5  let x = 1 in
6  let x' = set(x, 1) in
7
8  let a = greater_than_five x in
9  let b = greater_than_five x' in
10 let c = greater_than_five (x + x')
```

The code on lines 2 and 3 merely reads as "*if the expression that has been used to construct the value of x has been annotated with additional data of the type int, then bind this data to the name 'n' and compile itself into the first branch, or to the second, otherwise*", by which we instruct the compiler how to interpret the case when it finds out that the expression has some data has been associated with it. In our case, we tell it to use the data associated data and perform computation with the data rather than with the real argument that shall be evaluated at runtime.

But now it does not seems to be so straightforwards how to implement this. The thing is, the function's body is intended to be evaluated at runtime, but we promised that we handle all static annotations at the stage of compilation. On the

other hand, lifting the construction's evaluation to the compile-time prevents us from compiling functions ahead: the function's body for calls on lines 8 and 9 must be different.

As the investigation of the problem requires some understanding of our approach, it seems that it is the very time to start discussing how we can use types to handle static annotations and then return to the problem later.

2.2 Implementation

Using types to keep, retrieve, and propagate annotations has definitely something to do with the type of $set(e, v)$; at least because one is responsible for setting them. As a suggestion, we can impose its type, for instance, to keep the associated data and e's type within itself:

$$\frac{\Gamma \vdash x : \text{Int} \qquad \Gamma \vdash 1 : \text{Int}}{\Gamma \vdash \text{set}(x, 1) : [\text{Int}]\langle 1 : \text{Int}\rangle}$$

To not confuse readers, let us return to the first example and ascribe some expressions their corresponding types in comments:

```
1  let x = 1 in # x : Int
2  let x' = set(x, 1) in # x' : [Int] (a : Int)
3  x + x' + get(x') # ???
```

Now, as we can see, the only thing the construction $get(x')$ needs to do is to introspect the type of x' and substitute itself with the data it acquires. Its typing rule may be defined as follows:

$$\frac{\Gamma \vdash e_1 : [T_1](e_2 : T_2)}{\Gamma \vdash get(e_1) : T_2}$$

It seems that using types in such a way solves the first two problems: we are now able to save and retrieve these associations. Hopefully, it does; but now we are in troubles: the expression $x + x' + get(x')$ is not well-typed anymore. The reason for that is because now the type of x' is not integer, but rather the type [Int](1 : Int). However, such expression is indeed valid since, before we run a program, x' becomes an ordinary integer variable.

From now on, such types, which are created to handle static annotations, will be called *propertied types*, the expression we annotate data to (in our example, the variable x) will be called its *underlying expression*, and we will refer to the underlying expression's type as to the *base type* of a propertied one. As an example, in $e : T_e, v : T_v \vdash set(e, v) : [T_e](v : T_v)$: the type $[T_e](v : T_v)$ is a propertied type based on T_e, whose underlying expression is e. The v, with respect to this propertied type, will be called its property. Moreover, as one may notice, a propertied types is just a type parametrized by values, except that it is anonymous type that also stores its underlying expression type and is created to handle static annotations (and, as we will see in Section 3, follows a specific structure to be more general than its this section's version). We can thus speak about propertied types as just of the special case of dependent ones.

Now, when we have established the terminology, we can return to the problem. To fix it, we can simply force our arithmetic operators to work on propertied types as on ordinary integer ones since, again, we know that they will be compiled into their underlying expressions before they go to the runtime. Here are the rules for the operator +:

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}}$$

$$\frac{\Gamma \vdash e_1 : [\text{Int}]\langle \cdots \rangle \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}}$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : [\text{Int}]\langle \cdots \rangle}{\Gamma \vdash e_1 + e_2 : \text{Int}}$$

$$\frac{\Gamma \vdash e_1 : [\text{Int}]\langle \cdots \rangle \qquad \Gamma \vdash e_2 : [\text{Int}]\langle \cdots \rangle}{\Gamma \vdash e_1 + e_2 : \text{Int}}$$

Now it is the very time to back to our problem from the previous section's example. Recall its code is:

```
1  func greater_than_five (x : Int) with
2    if-has x n : Int then n > 5
3    else x > 5 in
4
5  let x = 1 in
6  let x' = set(x, 1) in
7
8  let a = greater_than_five x in
9  let b = greater_than_five x' in
10 let c = greater_than_five (x + x')
```

And here we face another typing problem; this time, the ill-typed expression is *greater_than_five x'*. This is because, just as arithmetic operators, our function expects to receive arguments of the integer type, but at line 9, we pass it x', which is of a propertied one. To solve this, again, we need to force functions that work on the type X to receive arguments of propertied types based on X as well:

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{App-1}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : [T_1]\langle \cdots \rangle}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{App-2}$$

In other words, we make our functions *polymorphic with respect to propertied type*. As it turned out, proper implementation of the polymorphism solves the problem of correct propagation of static annotations from Section 2.1. That is, as long as we store all our annotations in a propertied type, propagating the type of an argument makes it possible for constructions *get* and, essentially, *if-has*, to acquire the data and, consequently, decide how to compile themselves.

As not all implementations of polymorphism satisfy our requirements, we now have to choose the appropriate one. For instance, we cannot follow the classical, i.e., "boxed", approach, in which bodies are compiled ahead, an argument is set to be of an unknown type and internal runtime checks are present; the presence of runtime checks is what makes it incompatible for us. An approach that seems to fit our requirements is what people usually call a "compile-time", "static", "early-binding", or "static-binding" polymorphism, depending on the language they came from. This approach mainly relies on the notion of *function monomorphization*, meaning that instead of boxing types and compiling functions ahead, we generate a specialized version of a function each time one is invoked and substitute that unknown type with the type of calling argument, so the resulting call is made to its specialized version. Indeed, if there is a specialized version that has the same body as we need, a new version shall not be generated; the resulting call shall just use it. Following this intuition, a moment before it goes to the runtime, we expect our program to look like the following:

```
1  func greater_than_five_1 (x : Int) with x > 5 in
2  func greater_than_five_2 (x : Int) with 1 > 5 in
3
4  let x = 1 in
5  let x' = x in
6
7  let a = greater_than_five_1 x in
8  let b = greater_than_five_2 x' in
9  let c = greater_than_five_1 (x + x')
```

As we can see, we expressed a trivial optimization that propagates the value of x'. Before we go further, it is worth noting that our concept should never be used to express such optimizations; most modern compilers, without any notion of type properties, may effectively apply an optimization called *constant propagation*, reducing these whole constant expressions to their equivalent values at the stage of compilation, and obtain even a better result. Propagating values of constant expressions in this way rather consumes a discernible amount of a compiler's resources and bloats the output binary file's size. We have chosen it just as a simple example and primarly to the reason that expressing something like linear types using this concept in an elegant manner is a whole separate research topic.

Nevertheless, sometimes, such optimizations in the code itself may be beneficial. As an example, we can annotate an array with its size hiddenly, provided the one is, of course, known. After that, in functions working on arrays, just as we did in the example above, we can instruct the compiler that if there is some data associated with an array that

represents its size, then use this data to lift some runtime checks to compile-time. Another such specimen is annotating instances of 'std::any', whether from the C++'s or Rust's world, with their inhabited types. Both cases, in fact, would consume a significant amount of the compiler's resources but will make it possible to lift 100% of runtime checks that can be made at the stage of compilation to the compile-time, without bloating the output binary's size when used appropriately, which can be beneficial for performance-critical applications that are compiled rarely.

2.3 Transformation

This is the very time to discuss how exactly we evaluate all these static annotations during compilation. This is the last part of this section, proceeding after which the reader shall be ready for the presentation of our type theory, where we formalize type properties.

As we said many times, we must handle all these annotations before we run a program. This means that, right before we pass it to the runtime, it must proceed through an additional phase where all these annotations are handled. We call this phase *transformation* since it is the standard name for phases introduced for partially evaluating programs before one is run.

Formally, we can express transformation as the judgment $\Gamma \vdash e_1 \twoheadrightarrow e_2 : T$, which merely reads as "e_1, *under the typing context* Γ*, is transformed into* e_2*, and the type of* e_2 *is* T". However, we also need to monomorphize our functions, which means that we are supposed to somewhere store their both ordinary and specialized versions. For this purpose, we introduce a new context on which we operate — the functional context Δ:

$$\Delta ::= \ | \ \Delta, f :: x : T_1 \,.\, M : T_2 \mid \Delta, f\,[n] \triangleright x : T_1 \,.\, M : T_2$$

The entry $f :: x : T_1 \,.\, M : T_2$ keeps ordinary user functions (such as *greater_than_five*), while $f\,[n] \triangleright x : T_1 \,.\, M : T_2$ designates f's specialized version (such as *greater_than_five_1* and *greater_than_five_2*). n here is just a natural number to distinguish different specialized versions.

Now our transformation shall look like the following:

$$\Gamma \vdash \langle \Delta_1; e_1 \rangle \twoheadrightarrow \langle \Delta_2; e_2 \rangle : T$$

which now also means that to pass the transformation, it's required to have the functional context of Δ_1, which, after the transformation, becomes Δ_2.

Just as the usual typing judgment works under a typing context, the same way the transformation judgments tend to work under assumptions about transformations of different parts of its program. For this purpose, as long as the transformation works under the typing context and is actually typed itself, rather than introducing another context, we can store transformation judgments right in Γ:

$$\Gamma ::= \cdots \mid \langle \Delta; x \rangle \twoheadrightarrow \langle \Delta'; e \rangle : T$$

where the judgment $\langle \Delta; x \rangle \twoheadrightarrow \langle \Delta'; e \rangle : T$ means that the variable x is transformed into the expression e, and the context changes from Δ to Δ'.

Now let us attempt to design transformation for simple cases. Constants, for instance, have nothing to do with type properties, so we transform them into themselves:

$$\frac{}{\Gamma \vdash \langle \Delta; n \rangle \twoheadrightarrow \langle \Delta; n \rangle : \mathrm{Int}}$$

where n is an integer constant.

Now it is the time to give a turn for *set* and *get*. As we said in previous sections, expressions of propertied type are transformed into their underlying expressions; we, however, have not yet defined what is a canonical term of a propertied type itself. As a suggestion, since propertied types serve only to hang on some additional data on an expression, and we store both the additional data and the expression's type in the propertied type itself, we could consider making any underlying expression be its canonical term. But this would break the uniqueness of typing for all other expressions, so we could instead uniformly denote $propertied[e]$ to be a canonical term of any propertied type, whose underlying expression is e.

Following this intuition, *set* has no choice but to be transformed into a proper term of a propertied type:

$$\frac{\Gamma \vdash \langle \Delta; e \rangle \twoheadrightarrow \langle \Delta_1; e' \rangle : T \qquad \Gamma \vdash \langle \Delta_1; v \rangle \twoheadrightarrow \langle \Delta'; v' \rangle : P}{\Gamma \vdash \langle \Delta; \text{set}(e, v) \rangle \twoheadrightarrow \langle \Delta'; \text{propertied}[e'] \rangle : [T](v' : P)}$$

The transformation of get is simple and we can transform it as follows:

$$\frac{\Gamma \vdash \langle \Delta; M \rangle \twoheadrightarrow \langle \Delta'; \text{propertied}[e'] \rangle : [T](v : P)}{\Gamma \vdash \langle \Delta; \text{get}(M) \rangle \twoheadrightarrow \langle \Delta'; v \rangle : P}$$

Thus, having defined transformation for simple cases, we can now switch on slightly more difficult ones. For example, here is how we can design the transformation for the expression let:

$$\frac{(T_1 \neq [T]\langle \cdots \rangle) \qquad \Gamma \vdash \langle \Delta; M \rangle \twoheadrightarrow \langle \Delta_M; M' \rangle : T_1 \qquad \Gamma, \langle \Delta; x \rangle \twoheadrightarrow \langle \Delta_M; x \rangle : T_1 \vdash \langle \Delta_M; N \rangle \twoheadrightarrow \langle \Delta_N; N' \rangle : T_2}{\Gamma \vdash \langle \Delta; \text{let } x = M \text{ in } N \rangle \twoheadrightarrow \langle \Delta_N; \text{let } x = M' \text{ in } N' \rangle : T_2}$$

$$\frac{\begin{array}{c}\Gamma, \Gamma' \vdash \langle \Delta_M; N[x/y] \rangle \twoheadrightarrow \langle \Delta_N; N' \rangle : T_2 \\ \Gamma \vdash \langle \Delta; M \rangle \twoheadrightarrow \langle \Delta_M; \text{propertied}[e] \rangle : [T_1]\langle a : T_a \rangle \\ (\Gamma' \equiv \langle \Delta; x \rangle \twoheadrightarrow \langle \Delta; x \rangle : T_1, \langle \Delta; y \rangle \twoheadrightarrow \langle \Delta_M; \text{propertied}[x] \rangle : [T_1]\langle a : T_a \rangle)\end{array}}{\Gamma \vdash \langle \Delta; \text{let } x = M \text{ in } N \rangle \twoheadrightarrow \langle \Delta_N; \text{let } x = e \text{ in } N' \rangle : T_2}$$

which is, as opposed to simple cases above, defined by two distinct rules. Consider an expression let $x = M$ in N. For now, there are two cases whose transformation we shall treat differently. The first is when M is an ordinary expression and is not of a propertied typed. In this case, there is nothing special to do: under the assumptions that M is transformed into M', that x transforms into itself, and that N transforms into N', we can transform the expression into let $x = M'$ in N', taking in account all rules with composing typing and functional contexts. This is what the first rule deals with. The second rule, in turn, deals with the case when M is of a propertied type. In this case, we shall transform M into its underlying expression and properly propagate its information to the continuation. That is, we must propagate the propertied type but must not propagate the underlying expression since the construction let is here to abstract from concrete values.

Following these rules, we obtain that the example above is transformed into the following program:

```
1  let x = 1 in
2  let x' = x in
3  x + x' + 1
```

just as we expected.

Rules for transforming the expression $if - has$ follow the same strategy; they explore the type of their expression and are transformed into one of their branches. Transformation of the function definition is just the expansion of our functional context with one more function definition:

$$\frac{\Gamma, x : T_1 \vdash M : T_2 \qquad \Gamma, f : T_1 \to T_2, \vdash \langle \Delta, f :: x : T_1 \,.\, M : T_2; e \rangle \twoheadrightarrow \langle \Delta'; e' \rangle : T_e}{\Gamma \vdash \langle \Delta; \text{func } f\ (x : T_1) \text{ with } M \text{ in } e \rangle \twoheadrightarrow \langle \Delta'; e' \rangle : T_e}\ \text{R-Func}$$

Rules for the transformation of function application are where things get complicated. In short, we take out the body of the function, perform its transformation, place the acquired instantiation into the functional context, and replace the function in the former expression with its specialized version. The complicated thing is that there are four cases of function applications that we have to treat differently, i.e for which the transformation shall be different. Moreover, generating specialized versions of a function for exactly each function call may be inefficient since a specialized version that fits our requirements may be already present in the context. So, to build a comprehensive intuition on how to design such rules, too many notions from our type theory must be introduced. Therefore, it seems that it is the very time to start presenting our type theory.

3 The system $\lambda_{\twoheadrightarrow_p}$: Formalization of Type Properties

In this section, we present a simple type theory [2] augmented with the notion of type properties. We call this system $\lambda_{\twoheadrightarrow p}$.

3.1 Syntax

The concrete syntax of $\lambda_{\twoheadrightarrow p}$ is defined as follows:

(*typing contexts*)	Γ	::=	$\mid \Gamma, x : T$
(*types*)	T, P	::=	int $\mid$ unit $\mid T_1 \to T_2$
(*expressions*)	F, M, N, L, e, t	::=	func $f\ x : T$ with M in N $\mid$ let $x = e$ in L
			$\mid$ if-has $L\ p : T$ bind-as x in M else N
			$\mid$ extract(M) $\mid$ set(M, p, e) $\mid$ get(M, p)
			$\mid$ erase(M, p) $\mid M\ N \mid e_1 + e_2 \mid e_1 - e_2$
			$\mid x \mid n$
(*identifiers and variables*)	x, y, f, p		
(*integer constants*)	n, k		

The system has two base types — *int* and *unit* — and one compound — the function type. The system also encounters propertied types. Although they are its key feature, one must observe that they are not present in the syntax; this is because we attempted to hide the fact for a user that we use them to handle static annotations, since it must look like the compiler mysteriously handles static annotations on its own.

The expressions *if-has*, *set*, and *get* are those constructions from pseudo-code samples from Section 2 but are generalized versions of them; the crucial distinction is that now each construction takes an identifier as an additional argument, and its justification is as follows. In a real language, a user may want to store different data in propertied types for different purposes. For this reason, we now make it possible to store an arbitrary amount of data in propertied types, each marked with a distinct name.

In contrast with Section 2, two more expressions operate on propertied types. These are *extract* and *erase*. As one can infer from their names, the former extracts the underlying value of an expression of a propertied type, while the latter erases certain property of one.

3.2 Typing Rules

Below we present inference rules for typing contexts and type formations.

$$\frac{}{ctx}\ \text{Ctx-1}$$

$$\frac{\Gamma\ ctx \qquad \Gamma \vdash T \qquad (x \notin \Gamma)}{\Gamma, x : T\ ctx}\ \text{Ctx-2}$$

$$\frac{\Gamma\ ctx}{\Gamma \vdash \text{unit}}\ \text{F-Unit}$$

$$\frac{\Gamma\ ctx}{\Gamma \vdash \text{int}}\ \text{F-Int}$$

$$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash T_2}{\Gamma \vdash T_1 \to T_2}\ \text{F-Func}$$

$$\frac{\Gamma \vdash T_1}{\Gamma \vdash [T_1]\langle\rangle}\ \text{F-Prop-1}$$

[2] The theory was originally based on *Simply Typed Lambda Calculus* (...), but augmenting it with type properties eliminated almost all observable similarities.

$$\frac{\Gamma \vdash [T]\langle p_1 \hookrightarrow e_1[P_1], \cdots, p_n \hookrightarrow e_n[P_n]\rangle \qquad \Gamma \vdash e : P \qquad (p_1 \neq p \wedge \cdots \wedge p_n \neq p)}{\Gamma \vdash [T]\langle p_1 \hookrightarrow e_1[P_1], \cdots, p_n \hookrightarrow e_n[P_n], p \hookrightarrow e[P]\rangle} \text{ F-Prop-2}$$

Here, $[T]\langle p_1 \hookrightarrow e_1[P_1], \cdots, p_n \hookrightarrow e_n[P_n]\rangle$ represents a propertied type based on the type T and has properties in the form of $p \hookrightarrow e[P]$, where p is an identifier used to refer to the property, and e is the property's expression of the type P. The notation $(p_1 \neq p, \cdots, p_n \neq p)$ in the rule *F-Prop-2* indicates that p must not be already present in the properties.

As we go through this section, in our inference rules, we use some notations, some of which, with their corresponding meaning, we list below:

- We use $\Gamma \uplus \{\Gamma'\}$ to denote a context obtained by taking union of Γ and Γ'.
- We use $[T]\langle \cdots \rangle$ to represent any propertied type based on the type T.
- We use $T_1 \neq T_2$ to indicate that the type T_1 must not have the form of the type T_2. For instance, $T_1 \neq P_1 \rightarrow P_2$ states that T_1 must not be a function type.
- We use $T_1 \neq T_2$ to indicate that the type T_1 must be of the form of T_2. For instance, $T_1 \neq P_1 \rightarrow P_2$ states that T_1 must be a function type.
- We use $M[e_1/e_2]$ to designate the substitution of e_1 with e_2 in M.

Next, we give a turn for introduction rules:

$$\frac{\Gamma\ ctx}{\Gamma \vdash () : \text{unit}} \text{ I-Unit}$$

$$\frac{\Gamma\ ctx}{\Gamma \vdash n : \text{int}} \text{ I-Int}$$

$$\frac{\begin{array}{cc}\Gamma \uplus \{x : T_1\} \vdash M : T_2 & \Gamma \uplus \{f : T_1\} \rightarrow T_2 \vdash e : T_e \\ \Gamma \uplus \{x : [T_1]\langle\rangle\} \vdash M : T_2 & (T_2 \neq [T]\langle \cdots \rangle) \wedge (T_2 \neq P_1 \rightarrow P_2)\end{array}}{\Gamma \vdash \text{func } f\ x : T_1 \text{ with } M \text{ in } e : T_e} \text{ I-Func}$$

$$\frac{\Gamma \vdash M : T_1 \qquad \Gamma \vdash e : T_2 \qquad (T_1 \neq [P]\langle \cdots \rangle)}{\Gamma \vdash \text{set}(M, p, e) : [T_1]\langle p \hookrightarrow e[T_2]\rangle} \text{ I-Prop-1}$$

$$\frac{\Gamma \vdash M : [T]\langle p_1 \hookrightarrow e_1[P_1], \cdots, p_n \hookrightarrow e_n[P_n]\rangle \qquad \Gamma \vdash e : P \qquad (p_1 \neq p, \cdots, p_n \neq p)}{\Gamma \vdash \text{set}(M, p, e) : [T]\langle p_1 \hookrightarrow e_1[P_1], \cdots, p_n \hookrightarrow e_n[P_n], p \hookrightarrow e[P]\rangle} \text{ I-Prop-2}$$

$$\frac{\Gamma \vdash M : [T]\langle props_1, p \hookrightarrow e[P], props_2\rangle \qquad \Gamma \vdash e' : P'}{\Gamma \vdash \text{set}(M, p, e') : [T]\langle props_1, p \hookrightarrow e'[P'], props_2\rangle} \text{ I-Prop-3}$$

The notation $(T_2 \neq [T]\langle \cdots \rangle)$ in *I-Func* imposes a restriction that functions can't return values of propertied types. [3]

The rule *I-Prop-3* states that we can update a specific's property expression in a propertied type. In this rule, we write $props_1$ for all properties $p_1 \hookrightarrow e_1[P_1], \cdots, p_n \hookrightarrow e_n[P_n]$ that come before one with the name p and $props_2$ for all that come after.

We present rules that cover typing judgment of the remaining expressions below.

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \text{ E-App-1}$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : [T_1]\langle \cdots \rangle}{\Gamma \vdash t_1\ t_2 : T_2} \text{ E-App-2}$$

[3] Notice that functions can't accept them either. This is because, when defining a function, we explicitly denote its argument type. But as long as there is no conventional syntax for propertied types, a programmer cannot define a function that accepts values of propertied types only.

$$\frac{\Gamma \uplus \{x : T_1\} \vdash M : T_2 \qquad \Gamma \vdash N : T_1 \qquad (T_2 \neq [T]\langle\cdots\rangle) \wedge (T_2 \neq P_1 \to P_2)}{\Gamma \vdash \text{let } x = N \text{ in } M : T_2} \text{ E-Let}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ E-Plus}$$

$$\frac{\Gamma \vdash e_1 : [\text{int}]\langle\cdots\rangle \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ E-P-Plus-1}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : [\text{int}]\langle\cdots\rangle}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ E-P-Plus-2}$$

$$\frac{\Gamma \vdash e_1 : [\text{int}]\langle\cdots\rangle \qquad \Gamma \vdash e_2 : [\text{int}]\langle\cdots\rangle}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ E-P-Plus-3}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 - e_2 : \text{int}} \text{ E-Minus}$$

$$\frac{\Gamma \vdash e_1 : [\text{int}]\langle\cdots\rangle \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 - e_2 : \text{int}} \text{ E-P-Minus-1}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : [\text{int}]\langle\cdots\rangle}{\Gamma \vdash e_1 - e_2 : \text{int}} \text{ E-P-Minus-2}$$

$$\frac{\Gamma \vdash e_1 : [\text{int}]\langle\cdots\rangle \qquad \Gamma \vdash e_2 : [\text{int}]\langle\cdots\rangle}{\Gamma \vdash e_1 - e_2 : \text{int}} \text{ E-P-Minus-3}$$

$$\frac{\Gamma \vdash M : [T]\langle\cdots, p \hookrightarrow e[P], \cdots\rangle}{\Gamma \vdash \text{get}(M, p) : P} \text{ E-Get-Prop}$$

$$\frac{\begin{array}{c}\Gamma, L : T_L, \Gamma' \vdash T_x \qquad \Gamma, L : [T_L]\langle\rangle, \Gamma' \vdash N : T \\ \Gamma, L : [T_L]\langle p \hookrightarrow x[T_x]\rangle, \Gamma' \uplus \{x : T_x\} \vdash M : T \qquad (T_L \neq [T']\langle\rangle)\end{array}}{\Gamma, L : T_L, \Gamma' \vdash \text{if-has } L\ p : T_x \text{ bind-as } x \text{ in } M \text{ else } N : T} \text{ E-If-Has-1}$$

$$\frac{\begin{array}{c}\Gamma, L : [T_L]\langle props\rangle, \Gamma' \vdash T_x \qquad \Gamma, L : [T_L]\langle props \setminus p\rangle, \Gamma' \vdash N : T \\ \Gamma, L : [T_L]\langle props \setminus p, p \hookrightarrow x[T_x]\rangle, \Gamma' \uplus \{x : T_x\} \vdash M : T\end{array}}{\Gamma, L : [T_L]\langle props\rangle, \Gamma' \vdash \text{if-has } L\ p : T_x \text{ bind-as } x \text{ in } M \text{ else } N : T} \text{ E-If-Has-2}$$

$$\frac{\Gamma \vdash M : [T]\langle\cdots\rangle}{\Gamma \vdash \text{extract}(M) : T} \text{ E-Ext}$$

$$\frac{\Gamma \vdash M : [T]\langle props_1, p \hookrightarrow e[P], props_2\rangle}{\Gamma \vdash \text{erase}(M, p) : [T]\langle props_1, props_2\rangle} \text{ E-Erase}$$

The rules *E-If-Has-1* and *E-If-Has-2* represent typing rules for the expression *if-has*. The rule *E-If-Has-1* states that if one tries to check a property's presence even on expressions of non-propertied types, then it is still well-typed. This is crucial because the expression that we investigate may easily be a function's argument, the real type of which is not known until the function is monomorhpized. The reason that they require the code for the branch *else* to be well-typed when L is of an empty propertied type will be explored in Section 3.3.

The rule *E-App-2* is one of the main rules that make functions be polymorphic with respect to propertied types. Namely, it states that if a function accepts an argument of the type T_1, then it also accepts an argument of a propertied type based on T_1. The notation $props$
p represents the list of all properties of $props$ excluding the one with name p.

As already said in Section 2, we write $propertied[e]$ to a term of any propertied type with the underlying expression e. We express it formally by this rule:

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vdash [T]\langle props \rangle}{\Gamma \vdash \text{propertied}[e] : [T]\langle props \rangle}\ \text{I-Prop-Repr}$$

For now, to complete typing rules, we are only left with the structural rule that deduces variables from typing contexts. Namely, the *Var* rule:

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

3.3 Transformation

In this section, we formalize the transformation phase, the very phase where all static associations are handled, by presenting a typed big-step operational semantics for it.

Rules presented in this section do not explicitly assume that terms their operate on are well-typed. This is because, as we will see in Section 3.4, only well-typed programs may be passed to their runtime, so it would be meaningless to perform transformation on ill-typed ones.

Just as in Section 2, for the transformation judgment we introduce an additional context that keeps relevant information about functions, which we call a *functional context*. This context is defined in the same way as it is done in Section 2:

$$\Delta ::= \ |\ \Delta, f :: x : T_1 \,.\, M : T_2 \mid \Delta, f\,[n] \triangleright x : T_1 \,.\, M : T_2$$

That is, $f :: x : T_1 \,.\, M : T_2$ indicates that f is a function that takes x of the type T_1 as an argument, its body is M, and the resulting type is T_2. It stands for a function that contains a raw code (not transformed yet) and therefore cannot be propagated to a runtime call. $f\,[n] \triangleright x : T_1 \,.\, M : T_2$, where n is any natural number, stands for a monomorphized version of f, meaning that its body has already gone through transformation and it is ready to be called. n here stands just for distinguishing different monomorphizations of the function f.

Below we present rules for Δ context formation:

$$\frac{}{dctx}\ \text{Delta-Ctx-1}$$

$$\frac{\Delta\ dctx \qquad (f \notin \Delta)}{\Delta, f :: x : T_1 \,.\, M : T_2\ dctx}\ \text{Delta-Ctx-2}$$

$$\frac{\Delta\ dctx \qquad (f\,[n] \notin \Delta)}{\Delta, f\,[n] \triangleright x : T_1 \,.\, M : T_2\ dctx}\ \text{Delta-Ctx-3}$$

Functional contexts are intended to work under new transformation judgments, which itself is intended to work under a typing context. We therefore extend formation rules for the latter:

$$\Gamma ::= \ |\ \Gamma, x : T \mid \Gamma, \langle \Delta_1; x \rangle \twoheadrightarrow_p \langle \Delta_2; L \rangle : T$$

$$\frac{\Gamma\ ctx \qquad \Delta\ dctx \qquad \Delta'\ dctx \qquad \Gamma \vdash x : T \qquad \Gamma \vdash L : T \qquad (x_{\twoheadrightarrow_p} \notin \Gamma)}{\Gamma, \langle \Delta; x \rangle \twoheadrightarrow_p \langle \Delta'; L \rangle : T\ ctx}\ \text{Ctx-3}$$

To illustrate, the judgment $\Gamma \vdash \langle \Delta_1; t_1 \rangle \twoheadrightarrow_p \langle \Delta_2; t_2 \rangle : T$ merely means that the program t_1, under the functional context Δ, is transformed into the program t_2, the functional context became Δ', and the type of t_2 is T.

We extend the list of useful notations we will use in the rest of this paper:

- $f \notin \Delta$ — there is no entry $f :: x : T_1 \,.\, M : T_2$, where T_1 and T_2 match *arbitrary* types and M matches *any* term, in the context Δ.

- $f\,[n] \notin \Delta$ — there is no entry $f\,[k] \triangleright x : T_1\,.\,M : T_2$, where T_1 and T_2 match *arbitrary* types, M matches *any* term, and k matches the specific number n, in the context Δ.
- $x_{\twoheadrightarrow_p} \notin \Gamma$ — there is no entry $\langle \Delta_1; x\rangle \twoheadrightarrow_p \langle \Delta_2; L\rangle : T$, where Δ_1 and Δ_2 match any contexts, L matches any term, and T matches any type, in the context Γ.

We also extend structural rules that work on typing context to cover the transformation judgment as follows:

$$\frac{\Gamma\ ctx \qquad (\langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta'; L\rangle : T \in \Gamma)}{\Gamma \vdash \langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta'; L\rangle : T}\ \text{R-S-Red}$$

$$\frac{\Gamma\ ctx \qquad (\langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta'; L\rangle : T \in \Gamma)}{\Gamma \vdash x : T}\ \text{R-S-Var}$$

$$\frac{\Gamma\ ctx \qquad (\langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta'; L\rangle : T \in \Gamma)}{\Gamma \vdash L : T}\ \text{R-S-Term}$$

$$\frac{\Gamma \vdash \langle \Delta_1, \Delta_2; L\rangle \twoheadrightarrow_p \langle \Delta_1', \Delta_2'; L'\rangle : T_L}{\Gamma \vdash \langle \Delta_2, \Delta_1; L\rangle \twoheadrightarrow_p \langle \Delta_1', \Delta_2'; L'\rangle : T_L}\ \text{R-S-Exchange-Delta-1}$$

$$\frac{\Gamma \vdash \langle \Delta_1, \Delta_2; L\rangle \twoheadrightarrow_p \langle \Delta_1', \Delta_2'; L'\rangle : T_L}{\Gamma \vdash \langle \Delta_1, \Delta_2; L\rangle \twoheadrightarrow_p \langle \Delta_2', \Delta_1'; L'\rangle : T_L}\ \text{R-S-Exchange-Delta-2}$$

Now it is turn for the real transformation rules. Constants and literals have nothing to do with propertied types, so it would be reasonable to transform them into themselves:

$$\frac{\Gamma\ ctx \qquad \Delta\ dctx}{\Gamma \vdash \langle \Delta; ()\rangle \twoheadrightarrow_p \langle \Delta; ()\rangle : \text{unit}}\ \text{R-V-Unit}$$

$$\frac{\Gamma\ ctx \qquad \Delta\ dctx}{\Gamma \vdash \langle \Delta; n\rangle \twoheadrightarrow_p \langle \Delta; n\rangle : \text{int}}\ \text{R-V-Int}$$

$$\frac{\Gamma\ ctx \qquad \Delta\ dctx \qquad (f :: x : T_1\,.\,M : T_2 \in \Delta)}{\Gamma \vdash \langle \Delta; f\rangle \twoheadrightarrow_p \langle \Delta; f\rangle : T_1 \to T_2}\ \text{R-V-Func}$$

Next, we give turn to the expression *set*:

$$\frac{\Gamma \vdash \langle \Delta; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T \qquad \Gamma \vdash \langle \Delta_M; e\rangle \twoheadrightarrow_p \langle \Delta'; e'\rangle : P \qquad (T \neq [T']\langle\cdots\rangle)}{\Gamma \vdash \langle \Delta; \text{set}(M, p, e)\rangle \twoheadrightarrow_p \langle \Delta'; \text{propertied}[M']\rangle : [T]\langle p \hookrightarrow e'[P]\rangle}\ \text{R-Set-1}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta_M; e\rangle \twoheadrightarrow_p \langle \Delta'; e'\rangle : P \qquad (p_1 \neq p \wedge \cdots \wedge p_n \neq p) \\ \Gamma \vdash \langle \Delta; M\rangle \twoheadrightarrow_p \langle \Delta_M; \text{propertied}[M']\rangle : [T]\langle p_1 \hookrightarrow e_1[P_1], \cdots, p_n \hookrightarrow e_n[P_n]\rangle\end{array}}{\Gamma \vdash \langle \Delta; \text{set}(M, p, e)\rangle \twoheadrightarrow_p \langle \Delta'; \text{propertied}[M']\rangle : [T]\langle p_1 \hookrightarrow e_1[P_1], \cdots, p_n \hookrightarrow e_n[P_n], p \hookrightarrow e'[P]\rangle}\ \text{R-Set-2}$$

$$\frac{\Gamma \vdash \langle \Delta; M\rangle \twoheadrightarrow_p \langle \Delta_M; \text{propertied}[M']\rangle : [T]\langle props_1, p \hookrightarrow e_1[P_1], props_2\rangle \qquad \Gamma \vdash \langle \Delta_M; e_2\rangle \twoheadrightarrow_p \langle \Delta'; e_2'\rangle : P_2}{\Gamma \vdash \langle \Delta; \text{set}(M, p, e_2)\rangle \twoheadrightarrow_p \langle \Delta'; \text{propertied}[M']\rangle : [T]\langle props_1, p \hookrightarrow e_2'[P_2], props_2\rangle}\ \text{R-Set-3}$$

One more thing to note before we continue. One may observe that we do not evaluate the expression of the property but rather store its transformed version. A better approach would be to impose some restrictions, i.e., permit occurrence only of constant expressions. Then, given that we store only constant expressions, we can evaluate them already at the stage of compilation and do not store expressions but rather their concrete values. Although it is a great way to go, two reasons made us choose the other. The first thing to note is that all expressions in our type theory are constant expressions, and thus we can evaluate them at compile-time. The second is that this theory, being here only for demonstrational purposes, i.e., for the formalization of our concept, is already overcomplicated to be called simple,

meaning that bringing proper constant evaluation will bring the theory to another level of complication and cause even more confusion in the reader's eyes.

Next go the rules for retrieving and erasing type properties, as well as extracting their underlying values:

$$\frac{\Gamma \vdash \langle \Delta; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : [T]\langle \cdots, p \hookrightarrow e[P], \cdots\rangle \qquad \Gamma \vdash \langle \Delta_M; e\rangle \twoheadrightarrow_p \langle \Delta_e; e'\rangle : P}{\Gamma \vdash \langle \Delta; \text{get}(M, p)\rangle \twoheadrightarrow_p \langle \Delta_e; e'\rangle : P} \text{ R-Get}$$

$$\frac{\Gamma \vdash \langle \Delta; M\rangle \twoheadrightarrow_p \langle \Delta'; \text{propertied}[M']\rangle : [T]\langle \cdots\rangle}{\Gamma \vdash \langle \Delta; \text{extract}(M)\rangle \twoheadrightarrow_p \langle \Delta'; M'\rangle : T} \text{ R-Ext}$$

$$\frac{\Gamma \vdash \langle \Delta; M\rangle \twoheadrightarrow_p \langle \Delta'; \text{propertied}[M']\rangle : [T]\langle props_1, p \hookrightarrow e[P], props_2\rangle}{\Gamma \vdash \langle \Delta; \text{erase}(M, p)\rangle \twoheadrightarrow_p \langle \Delta'; \text{propertied}[M']\rangle : [T]\langle props_1, props_2\rangle} \text{ R-Erase}$$

Notice that the expression *get*, as opposed to *if-has*, works only on expressions of propertied types. This means that we are unable to retrieve a property from a function's argument without checking that it is present, since it is impossible for a function to work only on propertied types — they must work on their base types too.

$$\frac{\begin{array}{c}(T_L \neq [T']\langle \cdots\rangle) \wedge (y \notin \Gamma) \qquad \Gamma \vdash \langle \Delta; L\rangle \twoheadrightarrow_p \langle \Delta_L; L'\rangle : T_L \\ \Gamma, y : [T_L]\langle\rangle, \langle \Delta; y\rangle \twoheadrightarrow_p \langle \Delta_L; \text{propertied}[L']\rangle : [T_L]\langle\rangle \vdash \langle \Delta_L; N[L/y]\rangle \twoheadrightarrow_p \langle \Delta_N; N'\rangle : T_N\end{array}}{\Gamma \vdash \langle \Delta; \text{if-has } L\ p : T_x \text{ bind-as } x \text{ in } M \text{ else } N\rangle \twoheadrightarrow_p \langle \Delta_N; N'\rangle : T_N} \text{ R-If-Has-1}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta_L; N\rangle \twoheadrightarrow_p \langle \Delta_N; N'\rangle : T_N \qquad (p_1 \neq p, \cdots, p_n \neq p) \\ \Gamma \vdash \langle \Delta; L\rangle \twoheadrightarrow_p \langle \Delta_L; L'\rangle : [T]\langle p_1 \hookrightarrow e_1[P_1], \cdots, p_n \hookrightarrow e_n[P_n]\rangle\end{array}}{\Gamma \vdash \langle \Delta; \text{if-has } L\ p : T_x \text{ bind-as } x \text{ in } M \text{ else } N\rangle \twoheadrightarrow_p \langle \Delta_N; N'\rangle : T_N} \text{ R-If-Has-2}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta_L; N\rangle \twoheadrightarrow_p \langle \Delta_N; N'\rangle : T_N \qquad (P \neq T_x) \\ \Gamma \vdash \langle \Delta; L\rangle \twoheadrightarrow_p \langle \Delta_L; L'\rangle : [T]\langle \cdots, p \hookrightarrow e[P], \cdots\rangle\end{array}}{\Gamma \vdash \langle \Delta; \text{if-has } L\ p : T_x \text{ bind-as } x \text{ in } M \text{ else } N\rangle \twoheadrightarrow_p \langle \Delta_N; N'\rangle : T_N} \text{ R-If-Has-3}$$

$$\frac{\begin{array}{c}(P \neq T_1 \to T_2) \wedge (T_M \neq T_1 \to T_2) \wedge (T_M \neq [T_P]\langle \cdots\rangle) \\ \Gamma \vdash \langle \Delta_L; e\rangle \twoheadrightarrow_p \langle \Delta_e; e'\rangle : P \\ \Gamma \vdash \langle \Delta; L\rangle \twoheadrightarrow_p \langle \Delta_L; L'\rangle : [T]\langle \cdots, p \hookrightarrow e[P], \cdots\rangle \\ \Gamma \uplus \{x : P, \langle; x\rangle \twoheadrightarrow_p \langle; x\rangle : P\} \vdash \langle \Delta_e; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M\end{array}}{\Gamma \vdash \langle \Delta; \text{if-has } L\ p : P \text{ bind-as } x \text{ in } M \text{ else } N\rangle \twoheadrightarrow_p \langle \Delta_M; \text{let } x = e' \text{ in } M'\rangle : T_M} \text{ R-If-Has-4}$$

$$\frac{\begin{array}{c}(P = T_1 \to T_2) \vee (T_M = T_1 \to T_2) \vee (T_M = [T_P]\langle \cdots\rangle) \\ \Gamma \vdash \langle \Delta_L; e\rangle \twoheadrightarrow_p \langle \Delta_e; e'\rangle : P \\ \Gamma \vdash \langle \Delta; L\rangle \twoheadrightarrow_p \langle \Delta_L; L'\rangle : [T]\langle \cdots, p \hookrightarrow e[P], \cdots\rangle \\ \Gamma \uplus \{x : P, \langle \Delta_L; x\rangle \twoheadrightarrow_p \langle \Delta_e; e'\rangle : P\} \vdash \langle \Delta_e; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M\end{array}}{\Gamma \vdash \langle \Delta; \text{if-has } L\ p : P \text{ bind-as } x \text{ in } M \text{ else } N\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M} \text{ R-If-Has-5}$$

The rules above define the transformation of the expression *if-has*.

Recall that our properties store transformed expressions rather than their computed values. This is why, in *R-If-Has-4* and *R-If-Has-5*, we first compute this expression and assign it to the variable, rather than just substituting the identifier for this expression.

Furthermore, the rules enforce L enter the *else* branch being of a propertied type, even if the one is not. This is essential because, otherwise, the L's type with which it enters the branch would be unknown. This is because there are three cases in which we can send L to the *else* branch. The first is when L is of a propertied type but does not have the desired property (the rule *R-If-Has-2*). The second is when L is again of a propertied type, but the property we are looking for has a different type (the rule *R-If-Has-3*). The third is when L is not of a propertied type, meaning that it obviously lacks the property we are searching for (the rule *R-If-Has-1*); because L, in two cases above, enters the branch *else* being of a propertied type, we shall send it to the branch L being of a propertied type as well.

The transformation rule for a function definition expression is as follows:

$$\frac{\Gamma \uplus \{x : T_1\} \vdash M : T_2 \qquad \Gamma \uplus \{f : T_1 \to T_2\} \vdash \langle \Delta, f :: x : T_1 \,.\, M : T_2; e\rangle \twoheadrightarrow_p \langle \Delta'; e'\rangle : T_e}{\Gamma \vdash \langle \Delta; \text{func } f\ x : T_1 \text{ with } M \text{ in } e\rangle \twoheadrightarrow_p \langle \Delta'; e'\rangle : T_e}\ \text{R-Func}$$

The requirement on the typing judgment $\Gamma \uplus \{x : T_1\} \vdash M : T_2$ is only needed to infer the type T_2, not to ensure that the body is well-typed. As we already said, this is because it is meaningless to perform the transformation on an ill-typed program.

$$\frac{\begin{array}{c}(f\,[k] \notin \Delta) \\ (T_1 \neq [T]\langle\cdots\rangle) \wedge (T_1 \neq P_1 \to P_2) \qquad \Gamma \vdash \langle \Delta_i; F\rangle \twoheadrightarrow_p \langle \Delta, \Delta_f; f\rangle : T_1 \to T_2 \\ (\Delta_{f[k]} \equiv f\,[k] \triangleright x : T_1 \,.\, M' : T_2) \qquad \Gamma \vdash \langle \Delta_M, \Delta_f, \Delta_{f[k]}; N\rangle \twoheadrightarrow_p \langle \Delta'; N'\rangle : T_1 \\ (\Delta_f \equiv f :: x : T_1 \,.\, M : T_2) \qquad \Gamma \uplus \{x : T_1, \langle; x\rangle \twoheadrightarrow_p \langle; x\rangle : T_1\} \vdash \langle \Delta; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_2\end{array}}{\Gamma \vdash \langle \Delta_i; FN\rangle \twoheadrightarrow_p \langle \Delta'; f\,[k]\ N'\rangle : T_2}\ \text{R-App-Compile}$$

$$\frac{\begin{array}{c}(T_1 \neq [T]\langle\cdots\rangle) \wedge (T_1 \neq P_1 \to P_2) \\ (\Delta_{f[k]} \equiv f\,[k] \triangleright x : T_1 \,.\, M' : T_2) \\ (\Delta_f \equiv f :: x : T_1 \,.\, M : T_2) \\ \Gamma \vdash \langle \Delta_M, \Delta_f, \Delta_{f[k]}; N\rangle \twoheadrightarrow_p \langle \Delta'; N'\rangle : T_1 \\ \Gamma \vdash \langle \Delta_i; F\rangle \twoheadrightarrow_p \langle \Delta, \Delta_f, \Delta_{f[k]}; f\rangle : T_1 \to T_2 \\ \Gamma \uplus \{x : T_1, \langle; x\rangle \twoheadrightarrow_p \langle; x\rangle : T_1\} \vdash \langle \Delta; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_2\end{array}}{\Gamma \vdash \langle \Delta_i; FN\rangle \twoheadrightarrow_p \langle \Delta'; f\,[k]\ N'\rangle : T_2}\ \text{R-App-Ready}$$

$$\frac{\begin{array}{c}(f\,[k] \notin \Delta \cup \Delta_{f'}) \\ (\Delta_f \equiv f :: x : P_1 \to P_2 \,.\, M : T_M) \\ (\Delta_{f[k]} \equiv f\,[k] \triangleright x : P_1 \to P_2 \,.\, M' : T_M) \\ \Gamma \vdash \langle \Delta, \Delta_f; N\rangle \twoheadrightarrow_p \langle \Delta_{f'}, \Delta_f; f'\rangle : P_1 \to P_2 \\ \Gamma \vdash \langle \Delta_i; F\rangle \twoheadrightarrow_p \langle \Delta, \Delta_f; f\rangle : (P_1 \to P_2) \to T_M \\ \Gamma \uplus \{x : P_1 \to P_2, \langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta_{f'}; f'\rangle : P_1 \to P_2\} \vdash \langle \Delta_{f'}; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M\end{array}}{\Gamma \vdash \langle \Delta_i; FN\rangle \twoheadrightarrow_p \langle \Delta_M, \Delta_f, \Delta_{f[k]}; f\,[k]\ f'\rangle : T_M}\ \text{R-App-Compile-Func}$$

$$\frac{\begin{array}{c}(\Delta_f \equiv f :: x : P_1 \to P_2 \,.\, M : T_M) \\ (\Delta_{f[k]} \equiv f\,[k] \triangleright x : P_1 \to P_2 \,.\, M' : T_M) \\ \Gamma \vdash \langle \Delta_i; F\rangle \twoheadrightarrow_p \langle \Delta, \Delta_f; f\rangle : (P_1 \to P_2) \to T_M \\ \Gamma \vdash \langle \Delta, \Delta_f; N\rangle \twoheadrightarrow_p \langle \Delta_{f'}, \Delta_f, \Delta_{f[k]}; f'\rangle : P_1 \to P_2 \\ \Gamma \uplus \{x : P_1 \to P_2, \langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta_{f'}; f'\rangle : P_1 \to P_2\} \vdash \langle \Delta_{f'}; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M\end{array}}{\Gamma \vdash \langle \Delta_i; FN\rangle \twoheadrightarrow_p \langle \Delta_M, \Delta_f, \Delta_{f[k]}; f\,[k]\ f'\rangle : T_M}\ \text{R-App-Ready-Func}$$

$$\frac{\begin{array}{c}(f\,[k] \notin \Delta \cup \Delta_y) \wedge (T_1 \neq [P_1 \to P_2]\langle\cdots\rangle) \wedge (y \notin \Gamma) \\ (\Delta_f \equiv f :: x : T_1 \,.\, M : T_2) \\ (\Delta_{f[k]} \equiv f\,[k] \triangleright x : T_1 \,.\, M' : T_2) \\ (\Gamma_f \equiv x : T_1, y : [T_1]\langle props\rangle, \langle \Delta; y\rangle \twoheadrightarrow_p \langle \Delta_y; \text{propertied}[x]\rangle : [T_1]\langle props\rangle) \\ \Gamma \vdash \langle \Delta_i; F\rangle \twoheadrightarrow_p \langle \Delta, \Delta_f; f\rangle : T_1 \to T_2 \\ \Gamma \uplus \{\Gamma_f\} \vdash \langle \Delta_y; M[x/y]\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_2 \\ \Gamma \vdash \langle \Delta, \Delta_f; N\rangle \twoheadrightarrow_p \langle \Delta_y, \Delta_f; \text{propertied}[e]\rangle : [T_1]\langle props\rangle\end{array}}{\Gamma \vdash \langle \Delta_i; FN\rangle \twoheadrightarrow_p \langle \Delta_M, \Delta_f, \Delta_{f[k]}; f\,[k]\ e\rangle : T_2}\ \text{R-App-Compile-Prop-1}$$

$$\frac{\begin{array}{c}(T_1 \neq [P_1 \to P_2]\langle\cdots\rangle) \wedge (y \notin \Gamma) \\ (\Delta_f \equiv f :: x : T_1 \;.\; M : T_2) \\ (\Delta_{f[k]} \equiv f\,[k] \triangleright x : T_1 \;.\; M' : T_2) \\ (\Gamma_f \equiv x : T_1, y : [T_1]\langle props\rangle, \langle \Delta; y\rangle \twoheadrightarrow_p \langle \Delta_y; \text{propertied}[x]\rangle : [T_1]\langle props\rangle) \\ \Gamma \vdash \langle \Delta_i; F\rangle \twoheadrightarrow_p \langle \Delta, \Delta_f; f\rangle : T_1 \to T_2 \\ \Gamma \uplus \{\Gamma_f\} \vdash \langle \Delta_y; M[x/y]\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_2 \\ \Gamma \vdash \langle \Delta, \Delta_f; N\rangle \twoheadrightarrow_p \langle \Delta_y, \Delta_f, \Delta_{f[k]}; \text{propertied}[e]\rangle : [T_1]\langle props\rangle\end{array}}{\Gamma \vdash \langle \Delta_i; FN\rangle \twoheadrightarrow_p \langle \Delta_M, \Delta_f, \Delta_{f[k]}; f\,[k]\;e\rangle : T_2}\ \text{R-App-Ready-Prop-1}$$

$$\frac{\begin{array}{c}(f\,[k] \notin \Delta \cup \Delta_{f'}) \\ (\Delta_f \equiv f :: x : P_1 \to P_2 \;.\; M : T_M) \\ (\Delta_{f[k]} \equiv f\,[k] \triangleright x : P_1 \to P_2 \;.\; M' : T_M) \\ (\Gamma_f \equiv x : [P_1 \to P_2]\langle props\rangle, \langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta_{f'}; \text{propertied}[f']\rangle : [P_1 \to P_2]\langle props\rangle) \\ \Gamma \uplus \{\Gamma_f\} \vdash \langle \Delta_{f'}; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M \\ \Gamma \vdash \langle \Delta_i; F\rangle \twoheadrightarrow_p \langle \Delta, \Delta_f; f\rangle : (P_1 \to P_2) \to T_M \\ \Gamma \vdash \langle \Delta, \Delta_f; N\rangle \twoheadrightarrow_p \langle \Delta_{f'}, \Delta_f; \text{propertied}[f']\rangle : [P_1 \to P_2]\langle props\rangle\end{array}}{\Gamma \vdash \langle \Delta_i; FN\rangle \twoheadrightarrow_p \langle \Delta_M, \Delta_f, \Delta_{f[k]}; f\,[k]\,f'\rangle : T_M}\ \text{R-App-Compile-Prop-2}$$

$$\frac{\begin{array}{c}(\Delta_f \equiv f :: x : P_1 \to P_2 \;.\; M : T_M) \\ (\Delta_{f[k]} \equiv f\,[k] \triangleright x : P_1 \to P_2 \;.\; M' : T_M) \\ (\Gamma_f \equiv x : [P_1 \to P_2]\langle props\rangle, \langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta_{f'}; \text{propertied}[f']\rangle : [P_1 \to P_2]\langle props\rangle) \\ \Gamma \uplus \{\Gamma_f\} \vdash \langle \Delta_{f'}; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M \\ \Gamma \vdash \langle \Delta_i; F\rangle \twoheadrightarrow_p \langle \Delta, \Delta_f; f\rangle : (P_1 \to P_2) \to T_M \\ \Gamma \vdash \langle \Delta, \Delta_f; N\rangle \twoheadrightarrow_p \langle \Delta_{f'}, \Delta_f, \Delta_{f[k]}; \text{propertied}[f']\rangle : [P_1 \to P_2]\langle props\rangle\end{array}}{\Gamma \vdash \langle \Delta_i; FN\rangle \twoheadrightarrow_p \langle \Delta_M, \Delta_f, \Delta_{f[k]}; f\,[k]\,f'\rangle : T_M}\ \text{R-App-Ready-Prop-2}$$

$$\frac{(\Delta_{f[k]} \equiv f\,[k] \triangleright x : T_1 \;.\; M : T_2) \qquad \Gamma \vdash \langle \Delta, \Delta_{f[k]}; N\rangle \twoheadrightarrow_p \langle \Delta, \Delta_{f[k]}; N'\rangle : T_1}{\Gamma \vdash \langle \Delta, \Delta_{f[k]}; f\,[k]\;N\rangle \twoheadrightarrow_p \langle \Delta, \Delta_{f[k]}; f\,[k]\;N'\rangle : T_2}\ \text{R-App-Compiled}$$

The rules above are responsible for the transformation of a function application. As we discussed in Section 2, in our system, functions, at the stage of transformation are monomorhpized. Rules *R-App-Compile-** perform this monomorphization, provided that there is no suitable specialized version in a context yet. Rules *R-App-Ready-** handle cases when the appropriate function is already in the context, thus they do not generate its new version but rather compile the call to this specialized one.

Now let us investigate these rules. In our system, there are four different types of function applications that we should treat differently:

- The first is when an argument of a non-propertied and non-function type is applied to a function. This is the simplest case, and its transformation is as follows. Firstly, to perform monomorphization, we need to obtain the name of the target function to retrieve its body from the functional context. For this, we perform the transformation of the function itself. Then, under the assumption that our argument is transformed into itself, we transform the function's body. Finally, we perform transformation for the argument and then transform the whole expression into the call to a specialized version of the function. This is what the rules *R-App-Compile* and *R-App-Ready* do.
- The second is when a function is applied to another function. The difference with the first case is that, given that a function's name is required to perform the transformation, we cannot transform the non-argument function's body under the assumption that the argument is transformed into itself; it needs to be transformed into the name of the function in the argument position. This is what the rules *R-App-Compile-Func* and *R-App-Ready-Func* do.
- The third is when an argument of a propertied type the underlying expression of which is not a function is applied to a function. In this case, we need to transform its body under the assumption that the argument is of the propertied type, not of its original one. As long as the underlying expression is not a function, we

can abstract over them, i.e., say that the underlying expression is an arbitrary variable. This is what the rules *R-App-Compile-Prop-1* and *R-App-Ready-Prop-1* do.

- The last case is when an argument of a propertied type the underlying expression of which is a function applied to another function. This case is different from the previous one in the same way as the first is different from the second. The rules *R-App-Compile-Prop-1* and *R-App-Ready-Prop-1* handle this case.

The last rule, *R-App-Compiled*, transforms an already transformed function call. Since the function call is already transformed, this rule transforms one to itself. Below we present rules for transformation of the *let* expression:

$$\frac{\begin{array}{c}(T \neq P_1 \to P_2) \wedge (y \notin \Gamma) \\ (\Gamma_f \equiv x : T, y : [T]\langle props\rangle, \langle \Delta; y\rangle \twoheadrightarrow_p \langle \Delta_N; \text{propertied}[x]\rangle : [T]\langle props\rangle) \\ \Gamma \vdash \langle \Delta; N\rangle \twoheadrightarrow_p \langle \Delta_N; \text{propertied}[e]\rangle : [T]\langle props\rangle \\ \Gamma \uplus \{\Gamma_f\} \vdash \langle \Delta_N; M[x/y]\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M\end{array}}{\Gamma \vdash \langle \Delta; \text{let } x = N \text{ in } M\rangle \twoheadrightarrow_p \langle \Delta_M; \text{let } x = e \text{ in } M'\rangle : T_M}\ \text{R-Let-Prop-1}$$

$$\frac{\Gamma \vdash \langle \Delta; N\rangle \twoheadrightarrow_p \langle \Delta_N; \text{propertied}[f]\rangle : [P_1 \to P_2]\langle props\rangle \qquad \Gamma \uplus \{\Gamma_M\} \vdash \langle \Delta_N; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M \\ (\Gamma_M \equiv x : [P_1 \to P_2]\langle props\rangle, \langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta_N; \text{propertied}[f]\rangle : [P_1 \to P_2]\langle props\rangle)}{\Gamma \vdash \langle \Delta; \text{let } x = N \text{ in } M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M}\ \text{R-Let-Prop-2}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta; F\rangle \twoheadrightarrow_p \langle \Delta_F; f\rangle : T_1 \to T_2 \\ \Gamma \uplus \{x : T_1 \to T_2, \langle \Delta; x\rangle \twoheadrightarrow_p \langle \Delta_F; f\rangle : T_1 \to T_2\} \vdash \langle \Delta_F; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M\end{array}}{\Gamma \vdash \langle \Delta; \text{let } x = F \text{ in } M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M}\ \text{R-Let-Func}$$

The rest of the rules are those which don't transform/evaluate their expressions but rather propagate the transformation of their subexpressions:

$$\frac{\Gamma \vdash \langle \Delta; e_1\rangle \twoheadrightarrow_p \langle \Delta_{e_1}; e_1'\rangle : \text{int} \qquad \Gamma \vdash \langle \Delta_{e_1}; e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; e_2'\rangle : \text{int}}{\Gamma \vdash \langle \Delta; e_1 + e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; e_1' + e_2'\rangle : \text{int}}\ \text{R-P-Plus}$$

$$\frac{\Gamma \vdash \langle \Delta; e_1\rangle \twoheadrightarrow_p \langle \Delta_{e_1}; e_1'\rangle : \text{int} \qquad \Gamma \vdash \langle \Delta_{e_1}; e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; e_2'\rangle : \text{int}}{\Gamma \vdash \langle \Delta; e_1 - e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; e_1' - e_2'\rangle : \text{int}}\ \text{R-P-Minus}$$

$$\frac{\begin{array}{c}(T \neq [T']\langle \cdots\rangle) \wedge (T \neq P_1 \to P_2) \\ \Gamma \vdash \langle \Delta; N\rangle \twoheadrightarrow_p \langle \Delta_N; N'\rangle : T \\ \Gamma \uplus \{x : T, \langle ; x\rangle \twoheadrightarrow_p \langle ; x\rangle : T\} \vdash \langle \Delta_N; M\rangle \twoheadrightarrow_p \langle \Delta_M; M'\rangle : T_M\end{array}}{\Gamma \vdash \langle \Delta; \text{let } x = N \text{ in } M\rangle \twoheadrightarrow_p \langle \Delta_M; \text{let } x = N' \text{ in } M'\rangle : T_M}\ \text{R-P-Let}$$

And, finally, the rules that make ordinary expressions that work on base data types work on propertied ones:

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta; e_1\rangle \twoheadrightarrow_p \langle \Delta_{e_1}; \text{propertied}[L_1]\rangle : [\text{int}]\langle \cdots\rangle \\ \Gamma \vdash \langle \Delta_{e_1}; e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; \text{propertied}[L_2]\rangle : [\text{int}]\langle \cdots\rangle\end{array}}{\Gamma \vdash \langle \Delta; e_1 + e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; L_1 + L_2\rangle : \text{int}}\ \text{R-P-Plus-1}$$

$$\frac{\Gamma \vdash \langle \Delta; e_1\rangle \twoheadrightarrow_p \langle \Delta_{e_1}; \text{propertied}[L_1]\rangle : [\text{int}]\langle \cdots\rangle \qquad \Gamma \vdash \langle \Delta_{e_1}; e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; e_2'\rangle : \text{int}}{\Gamma \vdash \langle \Delta; e_1 + e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; L_1 + e_2'\rangle : \text{int}}\ \text{R-P-Plus-2}$$

$$\frac{\Gamma \vdash \langle \Delta; e_1\rangle \twoheadrightarrow_p \langle \Delta_{e_1}; e_1'\rangle : \text{int} \qquad \Gamma \vdash \langle \Delta_{e_1}; e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; \text{propertied}[L_2]\rangle : [\text{int}]\langle \cdots\rangle}{\Gamma \vdash \langle \Delta; e_1 + e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; e_1' + L_2\rangle : \text{int}}\ \text{R-P-Plus-3}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta; e_1\rangle \twoheadrightarrow_p \langle \Delta_{e_1}; \text{propertied}[L_1]\rangle : [\text{int}]\langle \cdots\rangle \\ \Gamma \vdash \langle \Delta_{e_1}; e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; \text{propertied}[L_2]\rangle : [\text{int}]\langle \cdots\rangle\end{array}}{\Gamma \vdash \langle \Delta; e_1 - e_2\rangle \twoheadrightarrow_p \langle \Delta_{e_2}; L_1 - L_2\rangle : \text{int}}\ \text{R-P-Minus-1}$$

$$\frac{\Gamma \vdash \langle \Delta; e_1 \rangle \twoheadrightarrow_p \langle \Delta_{e_1}; \text{propertied}[L_1] \rangle : [\text{int}]\langle \cdots \rangle \qquad \Gamma \vdash \langle \Delta_{e_1}; e_2 \rangle \twoheadrightarrow_p \langle \Delta_{e_2}; e_2' \rangle : \text{int}}{\Gamma \vdash \langle \Delta; e_1 - e_2 \rangle \twoheadrightarrow_p \langle \Delta_{e_2}; L_1 - e_2' \rangle : \text{int}} \text{ R-P-Minus-2}$$

$$\frac{\Gamma \vdash \langle \Delta; e_1 \rangle \twoheadrightarrow_p \langle \Delta_{e_1}; e_1' \rangle : \text{int} \qquad \Gamma \vdash \langle \Delta_{e_1}; e_2 \rangle \twoheadrightarrow_p \langle \Delta_{e_2}; \text{propertied}[L_2] \rangle : [\text{int}]\langle \cdots \rangle}{\Gamma \vdash \langle \Delta; e_1 - e_2 \rangle \twoheadrightarrow_p \langle \Delta_{e_2}; e_1' - L_2 \rangle : \text{int}} \text{ R-P-Minus-3}$$

3.4 Operational Semantics

In this section, we endow our theory with operational semantics that describes its runtime. When a program is well-typed and has passed the transformation phase, there is no more need to carry typing information in functions. We, therefore, introduce a new context that stores monomorphizated functions but without their typing information:

$$\Phi ::= \ | \ \Phi, f\,[n]\ :: x \otimes M$$

In order to ensure that the real program evaluation can be started *only* after the stuff with type properties have been carried out at compile-time, we give this Φ context to programs only after a successful transformation and type check:

$$\frac{\vdash t_1 : T \qquad \vdash \langle ; t_1 \rangle \twoheadrightarrow_p \langle \Delta; t_2 \rangle : T \qquad \Phi \equiv \{ f\,[n]\ :: x \otimes M \mid (f\,[n] \triangleright x : T_1 \,.\, M : T_2) \in \Delta \} \qquad (T \neq [T']\langle \cdots \rangle)}{\Phi \triangleright \langle ; t_2 \rangle} \text{ Ready}$$

The rule above states that if, under empty typing context, the program t_1 is well-typed, transformed into t_2, and t_2 is not a value of a propertied type, then the program t_2 is ready to be executed and is given a Φ-context. We call the program t_1 *well-transformed* if it satisfies the conditions above. The restriction with t_2 not being a value of a propertied type is required because we prohibit all that stuff with type properties to occur at runtime. In the next section, we will prove that this is the only source of them.

Our list of notations now will encounter one more — we will write $f\,[n] \notin \Phi$ to impose a requirement on a Φ context that one must not have a record $f\,[n]\ :: x \otimes M$, where $f\,[n]$ are exactly these f and n, while x and M match arbitrary argument names and bodies, respectively.

A context that will keep track of variables and their corresponding values is defined as follows:

$$\sigma ::= \ | \ \sigma, x \hookrightarrow v$$

Below we present rules for the context σ formation:

$$\frac{}{sctx} \text{ Sigma-Ctx-1}$$

$$\frac{\sigma\ sctx \qquad \triangleright v\ val \qquad (x \notin \sigma)}{\sigma, x \hookrightarrow v\ sctx} \text{ Sigma-Ctx-2}$$

and give a turn for Φ contexts:

$$\frac{}{pctx} \text{ Phi-Ctx-1}$$

$$\frac{\Phi\ pctx \qquad (f\,[n] \notin \Phi)}{\Phi, f\,[n]\ :: x \otimes M\ pctx} \text{ Phi-Ctx-2}$$

Obtaining a Φ context for an expression ensures that the expression is ready to process its computation, so every transition rule implicitly assumes that its source has one. Since the resulting expression, to proceed with the computation, must have the one too, we added a rule that does just that:

$$\frac{\Phi \triangleright \langle \sigma; e \rangle \qquad \Phi \triangleright \langle \sigma; e \rangle \longmapsto \langle \sigma'; e' \rangle}{\Phi \triangleright \langle \sigma'; e' \rangle} \text{ Phi-Preserve}$$

We also add two more expressions to the language that are not present in the language itself but which help us dropping or retrieving previous values of variables:

$$e ::= \cdots \mid \text{drop } x \text{ after } e \mid \text{retrieve } x = v \text{ after } e$$

These expressions must not be used by programmers. Closed values are evaluated by the following rules:

$$\frac{\Phi\ pctx}{\Phi \triangleright ()\ val}\ \text{V-Unit}$$

$$\frac{\Phi\ pctx}{\Phi \triangleright n\ val}\ \text{V-Int}$$

$$\frac{\Phi, f\ [n]\ ::\ x \otimes M, \Phi'\ pctx}{\Phi, f\ [n]\ ::\ x \otimes M, \Phi' \triangleright f\ [n]\ val}\ \text{V-Func}$$

And, finally, the transition rules are:

$$\frac{\Phi\ pctx \qquad \sigma, x \hookrightarrow v, \sigma'\ sctx}{\Phi \triangleright \langle \sigma, x \hookrightarrow v, \sigma'; x\rangle \longmapsto \langle \sigma, x \hookrightarrow v, \sigma'; v\rangle}\ \text{Var}$$

$$\frac{\Phi \triangleright \langle \sigma; e_1\rangle \longmapsto \langle \sigma'; e_1'\rangle}{\Phi \triangleright \langle \sigma; e_1\ e_2\rangle \longmapsto \langle \sigma'; e_1'\ e_2\rangle}\ \text{App-P-1}$$

$$\frac{\Phi \triangleright e_1\ val \qquad \Phi \triangleright \langle \sigma; e_2\rangle \longmapsto \langle \sigma'; e_2'\rangle}{\Phi \triangleright \langle \sigma; e_1\ e_2\rangle \longmapsto \langle \sigma'; e_1\ e_2'\rangle}\ \text{App-P-2}$$

$$\frac{\Phi, f\ [n]\ ::\ x \otimes M, \Phi' \triangleright v\ val \qquad (x \notin \sigma)}{\Phi, f\ [n]\ ::\ x \otimes M, \Phi' \triangleright \langle \sigma; f\ [n]\ v\rangle \longmapsto \langle \sigma, x \hookrightarrow v; \text{drop } x \text{ after } M\rangle}\ \text{App-1}$$

$$\frac{\Phi, f\ [n]\ ::\ x \otimes M, \Phi' \triangleright v\ val}{\Phi, f\ [n]\ ::\ x \otimes M, \Phi' \triangleright \langle \sigma, x \hookrightarrow v_{\text{prev}}; f\ [n]\ v\rangle \longmapsto \langle \sigma, x \hookrightarrow v; \text{retrieve } x = v_{\text{prev}} \text{ after } M\rangle}\ \text{App-2}$$

$$\frac{\Phi, g\ [k]\ ::\ y \otimes M_g, f\ [n]\ ::\ x \otimes M_f, \Phi'\ pctx \qquad \sigma\ sctx}{\Phi, g\ [k]\ ::\ y \otimes M_g, f\ [n]\ ::\ x \otimes M_f, \Phi' \triangleright \langle \sigma; f\ [n]\ g\rangle \longmapsto \langle \sigma; M_f\rangle}\ \text{App-With-Func-1}$$

$$\frac{\Phi, f\ [n]\ ::\ x \otimes M_f, g\ [k]\ ::\ y \otimes M_g, \Phi'\ pctx \qquad \sigma\ sctx}{\Phi, f\ [n]\ ::\ x \otimes M_f, g\ [k]\ ::\ y \otimes M_g, \Phi' \triangleright \langle \sigma; f\ [n]\ g\rangle \longmapsto \langle \sigma; M_f\rangle}\ \text{App-With-Func-2}$$

$$\frac{\Phi \triangleright \langle \sigma; n_1\rangle \longmapsto \langle \sigma'; n_1'\rangle}{\Phi \triangleright \langle \sigma; n_1 + n_2\rangle \longmapsto \langle \sigma'; n_1' + n_2\rangle}\ \text{Plus-P-1}$$

$$\frac{\Phi \triangleright n_1\ val \qquad \Phi \triangleright \langle \sigma; n_2\rangle \longmapsto \langle \sigma'; n_2'\rangle}{\Phi \triangleright \langle \sigma; n_1 + n_2\rangle \longmapsto \langle \sigma'; n_1 + n_2'\rangle}\ \text{Plus-P-2}$$

$$\frac{\Phi \triangleright n_1\ val \qquad \Phi \triangleright n_2\ val \qquad \sigma\ sctx \qquad (n_1 + n_2 = n)}{\Phi \triangleright \langle \sigma; n_1 + n_2\rangle \longmapsto \langle \sigma; n\rangle}\ \text{Plus}$$

$$\frac{\Phi \triangleright \langle \sigma; n_1\rangle \longmapsto \langle \sigma'; n_1'\rangle}{\Phi \triangleright \langle \sigma; n_1 - n_2\rangle \longmapsto \langle \sigma'; n_1' - n_2\rangle}\ \text{Minus-P-1}$$

$$\frac{\Phi \triangleright n_1\ val \qquad \Phi \triangleright \langle \sigma; n_2\rangle \longmapsto \langle \sigma'; n_2'\rangle}{\Phi \triangleright \langle \sigma; n_1 - n_2\rangle \longmapsto \langle \sigma'; n_1 - n_2'\rangle}\ \text{Minus-P-2}$$

$$\frac{\Phi \triangleright n_1\ val \qquad \Phi \triangleright n_2\ val \qquad \sigma\ sctx \qquad (n_1 - n_2 = n)}{\Phi \triangleright \langle \sigma; n_1 - n_2 \rangle \longmapsto \langle \sigma; n \rangle}\ \text{Minus}$$

$$\frac{\Phi \triangleright \langle \sigma; e \rangle \longmapsto \langle \sigma'; e' \rangle}{\Phi \triangleright \langle \sigma; \text{let } x = e \text{ in } M \rangle \longmapsto \langle \sigma'; \text{let } x = e' \text{ in } M \rangle}\ \text{Let-P}$$

$$\frac{\Phi \triangleright v\ val \qquad (x \notin \sigma)}{\Phi \triangleright \langle \sigma; \text{let } x = v \text{ in } M \rangle \longmapsto \langle \sigma, x \hookrightarrow v; \text{drop } x \text{ after } M \rangle}\ \text{Let-1}$$

$$\frac{\Phi \triangleright v\ val}{\Phi \triangleright \langle \sigma, x \hookrightarrow v_{\text{prev}}; \text{let } x = v \text{ in } M \rangle \longmapsto \langle \sigma, x \hookrightarrow v; \text{retrieve } x = v_{\text{prev}} \text{ after } M \rangle}\ \text{Let-2}$$

$$\frac{\Phi \triangleright \langle \sigma; e \rangle \longmapsto \langle \sigma'; e' \rangle}{\Phi \triangleright \langle \sigma; \text{drop } x \text{ after } e \rangle \longmapsto \langle \sigma'; \text{drop } x \text{ after } e' \rangle}\ \text{Drop-After-1}$$

$$\frac{\Phi \triangleright v\ val}{\Phi \triangleright \langle \sigma, x \hookrightarrow v_x; \text{drop } x \text{ after } v \rangle \longmapsto \langle \sigma; v \rangle}\ \text{Drop-After-2}$$

$$\frac{\Phi \triangleright \langle \sigma; e \rangle \longmapsto \langle \sigma'; e' \rangle}{\Phi \triangleright \langle \sigma; \text{retrieve } x = v \text{ after } e \rangle \longmapsto \langle \sigma'; \text{retrieve } x = v \text{ after } e' \rangle}\ \text{Retrieve-After-1}$$

$$\frac{\Phi \triangleright v\ val}{\Phi \triangleright \langle \sigma, x \hookrightarrow v_x; \text{retrieve } x = v_{\text{prev}} \text{ after } v \rangle \longmapsto \langle \sigma, x \hookrightarrow v_{\text{prev}}; v \rangle}\ \text{Retrieve-After-2}$$

4 Properties of $\lambda_{\twoheadrightarrow_p}$

In this section, we are going to explore some important properties of $\lambda_{\twoheadrightarrow_p}$. We start with properties of compile-time program transformation.

4.1 Program transformation properties

One of the important claims we made in previous sections was that all static annotations are handled at the stage of compilation and are, therefore, not present at runtime. The following lemma represents this claim in a formal manner.

Lemma 4.1 *If $\Gamma \vdash t_1 : T$ and $\Gamma \vdash \langle \Delta; t_1 \rangle \twoheadrightarrow_p \langle \Delta'; t_2 \rangle : T$, then either T is of the form $[T']\langle \cdots \rangle$, or no subexpression in t_2 is of a type $[T']\langle \cdots \rangle$. Moreover, for every function $f\ [n] \triangleright x : T_1\ .\ M : T_2$ in Δ', neither T_1 nor T_2 is of the form $[T']\langle \cdots \rangle$, nor type of any subexpression of M is of the form $[T']\langle \cdots \rangle$.*

We allow t_2 to be of a propertied type since the rule *Ready* won't give a Φ context for one, which is necessary to pass to the runtime.

Proof 4.1 *By induction on the derivation rules of the judgment $\twoheadrightarrow_p$ and the typing rules from Section 3.2.*

The proof that neither T_1 nor T_2 of $f\ [n] \triangleright x : T_1\ .\ M : T_2$ may be of the form $[T']\langle \cdots \rangle$ is immediate by the typing rule *I-Func*, which states that in order to form a function, the resulting type must not be a propertied one. T_1 a priori cannot be a propertied type, since there is no conventional syntax for them in the language.

For the rules that act on values — *R-V-Unit* and *R-V-Int* — the proof is immediate since they are transformed into themselves. The rules that act on the expression *set* make it be of type $[T']\langle \cdots \rangle$, meaning the proof is immediate too. Next, the rules *R-Get* and *R-Ext* transform their expressions into their subexpressions, which are, by induction, assumed to have the desired property.

The proof that the same holds for every function in the context Δ' is obtained by the induction on the rules that perform monomorphization and add them to functional contexts.

What's interesting are the rules *R-App-*-Prop-1*: when the argument of a function application is a propertied type's inhabitant, it is transformed into its underlying expression. Since the underlying expression cannot be of an inhabitant of propertied type, we eliminated the presence of one in function applications.

Proofs for the remaining rules follow the same structure and are obtained by using the same techniques; therefore, we omit them here for space reasons.

The next important property of the $\twoheadrightarrow_p$-transformation is that it must be deterministic, since none of our compile-time constructions is expected to do something that causes non-deterministic behavior.

Lemma 4.2 *If* $\Gamma \vdash \langle \Delta; t_1 \rangle \twoheadrightarrow_p \langle \Delta_1; t_2 \rangle : T$ *and* $\Gamma \vdash \langle \Delta; t_1 \rangle \twoheadrightarrow_p \langle \Delta_2; t_2' \rangle : T'$*, then* $\Delta_1 = \Delta_2$, $t_2 = t_2'$ *and* $T = T'$

We didn't define equality judgment for types, terms, and contexts, so what we mean is the ordinary syntactic one.

Proof 4.2 *By induction on the derivation rules of the judgment* $\twoheadrightarrow_p$.

The proof is similar to the proof of *Lemma 4.1*. For some cases, there is only one rule defining its transformation, so assuming that the condition holds for all subexpressions makes the proof for them. Other rules explore their subexpressions' structure so that no expression can be transformed by two rules simultaneously.

To illustrate, let us investigate the rules *R-Let-Prop-1*, *R-Let-Prop-2*, *R-Let-Func*, and *R-P-Let*. They all operate on an expression $let\ x = N\ in\ M$, but:

- *R-Let-Prop-2* works only when N is of a propertied type and the underlying value is of a function one.
- *R-Let-Prop-1* works only when N is of a propertied type too but the underlying value is *not* of a function type, thus covering all cases that *R-Let-Prop-2* excludes.
- *R-Let-Func* works only when N is transformed into a function.
- *R-P-Let* works only when N is neither transformed into a function nor to a propertied type's inhabitant.

thus mutually excluding each other.

4.2 Type soundness

Now, when we showed that the compile-time part has the desired properties, we are able to prove one of the most important properties of a programming language — the property of being *type sound*. The *type soundness* theorem states that if a program passes its programming language's type checker, then it is guaranteed that it has a well-defined behavior when executed. It consists of two theorems — the first, called the *progress* theorem, which states that if an expression e is well-typed (and, in our case, well-transformed under $\twoheadrightarrow_p$) then either it is already a value, or we can proceed with its computation. And the second, called the *preservation* theorem, which states that if an expression e has the type T and $\Phi \triangleright \langle \sigma; e \rangle \longmapsto \langle \sigma'; e' \rangle$, then e' is of type T too.

Lemma 4.3 (Progress) *If* $\Phi \triangleright \langle \sigma; e \rangle$*, then either* $\Phi \triangleright \langle \sigma; e \rangle \longmapsto \langle \sigma'; e' \rangle$ *or* $\Phi \triangleright e\ val$

Proof 4.3 *By induction on the rules given in Section 3.4*

The proof for expressions that represent values is immediate. Next, the key thing is to observe that the operational semantics is defined for all expressions, except ones that work with type properties, such as *set*, *get*, and so forth. Since we proved (Lemma 4.1) that one cannot occur at runtime, so the assumption is justified.

Lemma 4.4 (Preservation) *if* $\Gamma \vdash e : T$ *and* $\Phi \triangleright \langle \sigma; e \rangle \longmapsto \langle \sigma'; e' \rangle$*, then* $\Gamma \vdash e' : T$

Proof 4.4 *By induction on the typing rules from Section 3.2 and the rules given in Section 3.4*

5 Related work

Indeed, systems where these static annotations are expressible — have existed. Often, these are systems that were designed for (or just support) expression of other formal logic, usually permitting users to embed one into themselves. That being said, as far as we are able to express and embed a system that encounters static annotations (for instance, our $\lambda_{\twoheadleftarrow_p}$), these static annotations are indeed expressible there. A great example of such systems is the programming language Racket, whose fascinating main paradigm, Language-Oriented programming, encourages one to create a domain-specific language first and only then solve its problem within this language.

But what we are concerned about is that, both in these systems and in general, there was no separate concept that would pay attention to the expression of static annotations individually. Indeed, for systems like Racket, such a concept would

be absolutely worthless; their powerful metaprogramming systems are capable of expressing ones already, meaning that it would be pointless to treat them separately in this way.

However, the rest of the programming languages, which were not designed to express other formal systems, may find the presence of such a concept is beneficial. The thing is, it would be very practical to do tricks like expressing linear types as functionality in a separate program module, without the need to transfer all that complex metaprogramming systems for expression of other formal logic into already overcomplicated languages; and this is where the need in a separate, distinct concept that takes care of expressing static annotations individually (and thus does not augment a language with anything else) arises.

For the time being, we are not aware of any such concept except ours. Consequently, we are not aware of such a concept that addresses and undertakes this problem in the way we described in the introduction, i.e., primarily as a generalization of the mentioned type-theoretic pattern to provide more information about language constructions.

With this in mind, for now, it gives an impression that the closest notion to ours is Racket's syntax object properties [4]. Syntax objects themselves are, in fact, not related to our concept at all, but one may argue that, when speaking of them as of means to hang additional data on language constructions, accompanied with their properties, our concept resembles them. This is indeed true, but our addressing of the problem from the type-theoretic perspective is exactly what makes us different.

To say, for instance, in type theory, to provide more information about construction, we typically associate it with the construction in our contexts, thus making the information's presence implicit for programmers. A case in point is the same substructural type systems:

$$x \overset{1}{:} \text{Int},\ y \overset{1}{:} \text{Int} \vdash x + y \overset{\sigma}{:} \text{Int}$$

Following this, a programmer, in case it is not one who explicitly imposes usage restrictions, when constructs the expression $x + y$, is not aware that we hiddenly collect the information about variables' usage. The same holds for our concept; it looks like the compiler mysteriously remembers what data we associate with which expressions. This, however, cannot be said about syntax object properties: to associate data with an expression, one first needs to form its syntax object and then attribute some data. But from that moment, one no more works with the original expression — it works on the syntax object, which is a distinct first-class entity.

Another important attribute of type systems is that they usually define the static part of programming languages, meaning that supplement information is given meaning already at the stage of compilation. While this holds for our concept (it has been proven that our all static annotations are evaluated before runtime), syntax objects, being ordinary first-class entities, are not aware (and, consequently, do not impose any restrictions) of the phase to handle them at [5]

The last two discernible specifics of our concept is that we use types to express static annotations and that we fully formalized our approach by presenting a simple type theory.

As a result, when speaking of them as of means to provide more information about expressions, syntax objects and their properties are more similar to objects and their dynamic attributes from the Object-Oriented world than to our concept: just like we wrap up our expressions into syntax objects and assign properties to them, we can in the same manner wrap up ones into objects and assign dynamic attributes to them.

Thus, these distinctions make both concepts serve different purposes and their applications diverse: syntax objects and their properties are an astounding tool when working with macros, while our concept is a great solution when the problem of providing additional information is undertaken primarily from the type-theoretic perspective.

[4] A syntax object is a Racket's first-class entity used to represent a piece of Racket's source code. That is, along with some meta-data describing this piece of source code, it contains its syntactic presentation and allows one to store some additional fields, called its properties, for one's purposes.

[5] It can be argued, however, that Racket provides facilities to lift evaluation to a higher level, thus making it possible to evaluate syntax object properties at compile-time. But, first of all, this is the attribute of Racket's metaprogramming system, not syntax object properties themselves. Further, we can even say that we can play with its macro system to erase the distinctions and make syntax object properties similar to our concept. Indeed we can: this would be nothing but the expression of our concept in Racket; as it is designed to express other formal systems, there is nothing special about this case. We also already argued why the presence of such a distinct concept can be beneficial for other systems that do not support an expression of others.