Introducing TYP

A next generation programming language

Scott Klarenbach
PointyHat Software
scott@pointyhat.ca

Summary

Programming languages have continually evolved toward higher levels of abstraction. Typically, each new generation of languages parses expressions ever closer to those used naturally by the human programmer. This is accomplished by outsourcing to a compiler many details that for years had been manually specified. Naturally this results in a direct correlation between language abstraction and programmer productivity, as we have seen historically in the path from Assembly; to C; through to Java; and Python. Continued gains along these lines are crucial if programmers are to keep up with the demands of a world in which automation is an expected commodity. Additionally, as end users interact with more and more software agents of ever increasing sophistication, they too will demand elegant tools that allow for routine automation without the help of a professional programmer. It is possible that programmers in thirty years will consider current scripting languages, like Python, to be very low level indeed, and regard them the same cautious curiosity with which a young programmer today might view Assembly. What useful abstractions will future programmers enjoy that enable the type of productivity shift we've seen in the past fifty years? And which of those abstractions can we start to introduce today that will allow the evolution of languages to continue in the spirit that has taken us all the way from Assembler to Ruby?

"When someone says 'I want a programming language in which I need only say what I wish done,' give him a lollipop." *- Alan Perlis*

1. Introduction

TYP is a new programming language intended to dramatically improve developer productivity across a wide range of domains. TYP introduces novel abstractions designed to relieve the programmer of many common responsibilities imposed by modern programming languages. The result is a set of semantics that are closer to natural language, and therefore easier to reason about than traditional source code. TYP proposes a syntax elegant enough to allow source code to be used as system documentation, enabling business stakeholders; systems analysts; programmers; and customer support to reference and reason about the same common document. Ultimately, TYP provides a framework for automation in various domains that allows novice programmers and power users to produce enterprise quality software without the aid of an experienced programmer.

2. Target Audience

Although TYP provides many sophisticated features that enable a programmer to develop novel and interesting solutions, it is not yet intended as a replacement for general purpose programming languages such as Java or C. Rather, TYP is targeted toward novice programmers, system administrators, business analysts and power users who wish to quickly prototype, model and interact with new domains without incurring the overhead complexity of a general purpose programming language. In many ways, TYP is analogous to SQL: a domain specific language designed as a succinct wrapper around common yet complex underlying patterns.

Throughout this document, we will provide examples of TYP code to help illustrate various concepts. Since TYP is intended to be understood by anyone, examples are often all that is needed to convey a concept. For example, the uninitiated can usually infer what is meant by the following.

```
every friday :at 3pm :for the next 2 months
  $list any ?incomplete #timesheets :where the @laborer was ?not-sick then
  $email the @result as a #csv-file :to #accounting
```

3. Philosophy of a Language

3.1 Yet another programming language?

There are countless new languages appearing on the scene every month, with the vast majority of them gaining little, if any, traction. A new language must therefore provide an order of magnitude improvement within the key areas it targets. The cost of learning a language is high, and the risk of adopting one in a professional context is much higher still. TYP may ultimately fail to be adopted outside of a small group, but even so, its creators will enthusiastically use and support it for years to come. The reason is simple.

TYP was created as an internal tool for developing commercial enterprise software applications. The programmers that use TYP and the company that sponsors it do so because it provides a competitive advantage. The advantage is quantified using real metrics and justified by the cold realities of the free market. There is also the implicit advantage of developer happiness, thanks to the ease of expression fostered by TYP. This metric is hard to measure, but easy to feel.

A few fundamental ideas and beliefs have motivated the creation of TYP. These ideas came about while building software in the real world, with nearly every concept selected to eliminate a specific suffering encountered by developers.

3.2 Reading vs Writing

Programmers spend most of their time reading and reasoning about existing code rather than writing new code. Therefore, it seems unwise to optimize a language for conciseness and brevity of syntax since a language with dense constructs or clever syntactic shorthands becomes very difficult to understand at a glance. It serves as a barrier to entry. Experienced programmers must first be indoctrinated into the idiosyncrasies of the language before they can understand the code, and non-programmers are usually excluded completely. Consider also that most programmers under the age of forty can probably type as fast as they speak, and the value in reducing language tokens to cryptic shorthand becomes even more unclear. Code should be written for the reader, in the same way that a newspaper article is written for the reader. Whatever literary device will make the expression most clear is precisely the one that should be used. A programming language that does not focus on the *reader* of code is an inefficient one - no matter how long it takes a human to type, or a computer to execute.

3.3 Explicit vs Implicit

Any details of an implementation that are implicitly understood rather than explicitly specified impose a conceptual burden on the reader. That is not to say that implicit concepts should not be permitted in a programming language. Indeed, many of the most useful abstractions of TYP result from semantics that are implied, but it should be acknowledged that any implicit magic provided by a language, no matter how convenient, is a cost to be paid upfront by those hoping to understand the code. Therefore, a good rule of thumb is that a programming language should make implicit only those devices that are already present in the common language of the user. For example, consider the following two lines of TYP.

```
$archive /my/file.pdf :to /tmp/file.zip
$email the @result :to scott@pointyhat.ca
```

Disregard any special characters to which you haven't been introduced, and take a guess as to the meaning of **@result**. In other words, *what* is emailed to scott@pointyhat.ca? It is obviously the result of the previous operation - the archived pdf file. Lower level languages would have us first come up with a name for some variable, and then explicitly assign the result of the first operation to that variable so that we could refer to it later. TYP does this for us automatically, and yet without any introduction to the concept, most readers are likely to understand the expression because the implicitness is not novel, but rather a common convention of the English language. It is this sort of implicitness that a language should strive to provide - that which eliminates mundane tasks without requiring any new understanding on the part of the reader.

This may seem verbose, and indeed many experienced programmers will admonish the idea as heresy, but remember that TYP is designed to be both the source code *and* the documentation of a given system. It is therefore better to be explicit, since understanding dense logical concepts is much easier if the reader doesn't have to fill in the blanks along the way. The following example shows this idea taken to extremes.

```
;; a novice TYP programmer would prefer this
$divide @net-profit :by @total-revenue then $multiply :by 100

;; to this
@net-profit / @total-revenue X 100

;; while a more experienced TYP programmer might state
@net-profit as a #percentage :of @total-revenue
```

Business applications do not have complicated mathematics, and so do not require a notation system that resembles calculus. The first style of code would certainly become unwieldy if used to desribe Newton's laws of motion, but there are already many great languages to use if that is your intention. The difference between the first and second samples above may seem trivial, and if you're an accountant you may even prefer the second. But remember, understanding any interesting system can not be done in isolation. Even for experienced programmers, parsing the second variety of code - dozens of lines at a time - is more straining, since it is less familiar than the more verbose version. TYP source code mimics the way a concept might be expressed around a boardroom table, and therefore results in no loss of fidelity when translated from the whiteboard to the keyboard.

3.4 Source Code as Documentation

Programmers are currently the only group that can make use of a source code document, and there is plenty of evidence that even they cannot efficiently reason about the code after a certain period of time. The most relevant and authoritative artifact - indeed often the only artifact - of a given software system is the source code, but due to specialized syntax it is inaccessible by the majority of the stakeholders in a given domain. Worse still, general purpose programming languages provide such numerous avenues of expression that code is often confusing even when shared between programmers on the same project. A far better situation would be one in which customer service, project managers, sales staff, board members *and* programmers referenced the source code directly in order to reason about, and make changes to, the software system.

If programmer productivity is a goal, source code must be quickly understandable regardless of when or by whom it was written. Anything less results in a burden known as *legacy software*, which is one of the greatest sources of suffering for the modern programmer.

4. Core Concepts

TYP introduces a few built in constructs outlined here so that reader can more easily follow the subsequent examples. Each construct in TYP is specified with a unique prefix character, to aid the reader in quickly parsing the context of an expression, regardless of the order in which the terms are presented.

4.1 Types

TYP is composed of **types**, which can be thought of as nouns or things, as they represent the objects in the environment that a programmer will interact with. Types are always prefixed with the **'#'** character. Some common examples include *#client*, *#invoice*, *#pdf*, *#server* and *#bank-account*.

4.2 Properties

Each type can have multiple **properties**, which are attributes of the thing being specified. Properties are always prefixed with the **'@'** character. For example, the type *#client* might have the properties *@first-name* and *@last-name*.

4.3 Actions

Each type has associated **actions**. These are like verbs, in that they operate on types in order to perform some process or task. Actions are always prefixed with the **'$'** character. Some common actions might include *$delete*, *$archive*, *$close* and *$send*.

4.4 States

Each type likely has associated **states** of being. These are analogous to adjectives, since they act to further qualify a given type. States are usually defined in terms of a type's underlying properties, and they are always prefixed with the **?** character. For example, given the type *#client* with the property *@budget*, we might define a state *?important* to mean any *@budget* that is larger than $100,000. We can then refer to *?important #clients*, as a quick qualifying filter.

4.5 Parameters

Each action that operates on a type is likely to have one or more **parameters**. These can be thought of as adverbs, because they further qualify the behavior of an action. Parameters are always prefixed with the **':'** character. For example, the action *$email* of the type *#string* has a parameter *:to*, which indicates to whom the email should be sent, i.e.,

```
$email ''Wanna grab lunch?'' :to scott@pointyhat.ca
```

4.6 Comments and Keywords

Finally, anything prefixed with the **';'** character is a comment which is ignored by TYP. The only other terms encountered in this document are **keywords**, and will be introduced by example as we move along. TYP keywords, such as *if* and *when*, are never prefixed by a special character.

5. Key Abstractions

The abstractions that distinguish TYP from lower level languages like Python or Ruby are as follows. These constructs promote a more direct translation between the intended logic and the source code. The developer specifies "what" they'd like to accomplish, and TYP strives hard to take care of the "how".

5.1 Data Types

In addition to the familiar types of most languages, such as strings, numbers, dates, booleans, etc., many more types that are common to modern development scenarios are provided, such as email addresses; PDF documents; databases; phone numbers etc. TYP goes further than simply providing a library of extensible types by implicitly recognizing types based on patterns in the source text. These patterns, like types themselves, are extensible by the programmer. For example, consider the following expression.

```
$email /my/file.pdf :to scott@pointyhat.ca
```

TYP automatically casts "/my/file.pdf" as the type *#pdf-document*, and sets both the types and values of its *@name* and *@location* properties accordingly. Similarly, "scott@pointyhat.ca" was recognized as the type *#email-address*. Note this wasn't a string of text as in many other languages, but rather a first-class data-type recognized directly by the language. Pattern matching of source text to data types provides the convenience of not having to explicitly state each type, and aids the source code in its utility as documentation; implicit, contextual typing is a common device of the English language.

Pattern matching also facilitates an elegant form of polymorphism, whereby the same action will trigger different results depending on the types in the surrounding context.

```
$send c:\cute-kitten.jpg :to scott@pointyhat.ca
;; sends a document as an email attachment

$send "drinks after work?" :to 778-319-4280
;; sends a text string as an sms message
```

Date types leverage this pattern matching to eliminate another great source of developer suffering. The following lines all produce valid *#dates* in TYP.

```
next friday evening
3 weeks from-now
2 days ago
last month
```

The patterns that map to types are extensible by the programmer. There are numerous uses of such a feature. One such case that comes default with TYP is the ability to specify formatting by example, rather than prescriptive regular expressions. For example:

```
$format #dates :like May 1, 2003
```

Types can be used for patterns, and vice versa, which is useful for text parsing and extraction.

```
$extract #phone-numbers :from /some/file.txt
```

5.2 States and Properties

Types have properties, as is outlined in section 4.2. These can be thought of as attributes that describe the object in question, and act as higher level qualifiers of a type, often by combining simpler properties. States play an important role in many of the inherent conveniences found in TYP. In addition to programmer-defined states, TYP automatically creates states in response to various expressions initiated by the programmer. For example, TYP provides the inverse of any state automatically, so that once *?active* is defined for *#client*, you can ask:

```
is #client 3 ?inactive
>> Yes.
```

TYP also commonly associates various actions with states. So, defining the action *$archive* for *#invoice* allows us to:

```
$get #invoices ?archived yesterday
```

5.3 Filtering

Armed with a set of states for a given type, programmers can constrain and filter actions based on state. This works for simple query operations and complex commands alike.

```
$list ?important #clients
$suspend ?overdue #accounts
```

TYP is forgiving of many qualifying words that are used in English, treating them as whitespace. You may have noticed "the @result" used in previous sections; "the" is ignored by TYP, as are several other commonly used words. TYP also understands that speakers frequently switch between singular and plural tenses when referring to types.

```
$list any ?important #clients
$print every ?important #client
```

We can combine states and parameters in flexible ways, so that our code will read like English even when performing multiple tasks at once.

```
$suspend any ?overdue #account :that is ?not-active
```

5.4 Variable Declaration and Assignment

Declaring new variables is a common task in any programming language yet it is often the case that programmers are forced to declare a variable merely as a placeholder for the previous operation, so that it can be referenced in the next operation. Poorly named variables are a pervasive source of confusing and inelegant source code. A language should provide abstractions for the most common types of scenarios so as to reduce the number of custom variable and function declarations. You have already seen an example of this in the section 2.

```
$archive any ?inactive #account ?smaller-than $50,000
$email the @result :to scott@pointyhat.ca
```

@result is automatically bound in this context for us. We could similarly use @it to refer to the last property in the same expression, without having to repeat our self.

```
$set @factor :to 5 if #account is ?important otherwise $set @it :to 3
```

This ability improves the expressiveness of actions with multiple clauses, or actions with repeated chains of *if* branches. In a typical language one usually has no choice but to be redundant.

```
;; instead of
if (@age == 3 or @age == 4 or @age = 7)
;; just say
if @age is 3, 4 or 7
```

A related feature is the implicit binding that happens for multiple actions in the same expression.

```
$backup then $delete every #pdf :in /this/directory

;; or how about

$list every ?maintenance #contract ?scheduled-for this-month
$log the @result :to /some/file.txt then $email the @number :to scott@pointyhat.ca
```

In the first example, each *#pdf* is first backedup, then deleted, just as one would expect. The second example is more dense, but if TYP is of any value, most readers will easily understand its purpose. The *@result* that is logged to /some/file.txt is the list of *?maintenance #contracts* that are *?scheduled-for* this month. Of greater interest is the second clause of the second expression.

```
$email the @number :to scott@pointyhat.ca
```

What is the *@number*? It is a property of *#contract*, and TYP knows this implicitly from the surrounding context even though it was two expressions prior (the logging happened in between).

Naturally there are many situations in which custom variable declaration is appropriate. TYP allows this in the traditional manner, but offers one additional method that fits more comfortably in many scenarios. Variables are often an afterthought, whereby the programmer first constructs an expression and then realizes the need to refer to it by name in the subsequent expression(s). TYP allows the semantics of the language to mimic the thought train of the programmer, using the **>** operator.

```
$get ?overdue #books > @late-books
$return any @late-books :that-are ?completed
```

This operator is also useful in other situations, and acts similar to the redirection operator familiar to most unix administrators.

```
c:\file.txt > @my-text
```

5.5 Events

You have seen how ?states play an integral role both for #types and $actions. Another area where states and actions interoperate in a useful way is the eventing framework that TYP makes available for any declared states. Consider an action *$complete* for a type *#book*. Once this action has been defined for the *#book*, TYP attaches a state representing the new action. This allows a programmer to declare:

```
when #book is ?completed $return @it :to the #library
```

Subsequent code in the environment, will automatically trigger the event.

```
$complete #book :titled "Being and Nothingness"
;; book is returned to the library
```

TYP strives hard for consistency, so any concepts learned early on should be applicable to more advanced scenarios. Keeping that in mind, we should be able to use state constraints with events, much as we previously did with actions.

```
when ?important #account is ?closed $notify #accounting
```

Here, the event is only triggered if an *?important #account* is *$closed*, rather than for every account. The flexibility of ordering also holds, so we could just as easily

```
$notify #accounting when ?important #account is ?closed.
```

5.6 Conversions

A very common activity performed manually by programmers in lower-level languages is the manual conversion from one type to another. This happens so frequently it can obscure the true intention of a section of code. Whether converting from strings to numbers, json to xml, or html to pdfs, programmers are forced to manually provide boiler-plate code to handle these conversions; however, when describing how a system should work at the conceptual level, most people take for granted that these conversions should occur, as is evident by the shorthand we inject into our everyday language. *"Send the invoice to the supplier"*. What is really meant is: *"Convert the invoice to a pdf file, attach the pdf file to an email, and then send the EMAIL to the supplier."* The conversion steps are implicit, and taken for granted in our speech. A programming language should work the same way.

TYP will search for any conversion paths that are present in the system to try and logically transform from one type to another. This happens without the programmer having to specify the path explicitly, regardless of the number of conversions required to satisfy the action. TYP comes with many conversions built in automatically, even for less than obvious scenarios.

```
$save www.newyorktimes.com/article :to /some/file.doc
;; implictly convert html -> xml -> MS Word
```

Of course, conversions are extensible so that programmers can define their own in a given domain so that they might state their intent in more natural language.

```
$invoice #supplier 1 :for #job 2

$email any ?new #clients as #xml :to owner@acme.com
```

5.7 Errors

Although programmers can explicitly instruct TYP in the event of an error, it is usually not necessary since TYP actively tries to prevent and resolve errors that might otherwise occur in lower level languages. In the event that an error does arise, TYP presents it to the programmer at a higher level of abstraction, usually as a derivative of some TYP construct such as a non-existent conversion. In much the same way as virtual machines liberated programmers from memory management, TYP provides several algorithms that implement best practice error handling techniques across a wide range of scenarios, at the language level. This can easily be configured by experienced programmers; however novices will benefit from a language that automates much of the error handling code that would otherwise have to be written by hand in a language like Javascript or Python.

5.8 Syntactic Extension

TYP is equipped with a powerful aliasing mechanism to allow for easy extension of the language. In the simplest realms, programmers can use the *$alias* action to tweak the types and keywords of the language in order to provide for more personalized expression.

```
$alias when :with whenever
;; and now we can say
whenever a ?new #lead is ?created
$schedule a 2pm #reminder :to ''Followup with @lead-name, @lead-phone-number''
```

More advanced pattern based aliasing is also possible. Recall the natural semantics of *:that* when filtering.

```
$list #clients :that are ?inactive
```

The parameter *:that* is merely a patterned alias for *:where ?state is @*. Moreover, states like *?inactive* are defined as patterned aliases of simpler properties, such as *not @active* or *@active is false*. Much of TYP is constructed in a similar manner; what is meant when we say TYP is written mostly in itself.

Aliases play a big role in the interoperability between types and actions as well.

```
$send #email "hey" :to scott@pointyhat.ca
;; equates to
$email "hey" :to scott

$create #archive :of /some/directory
;; is the same as
$archive /some/directory
```

Just as type automatically provides events for the states that we define, it also provides many aliases between actions and the types.

6. Real World Examples

The following section displays a few additional examples of how could be used in the real world.

```
;; system administration tasks
$define @dev-db :as #database :at localhost :named "my-project"
$define @production-db :as #database :at 192.168.0.100 :named "my-project"
tonight :at midnight
        $backup the @production-db :to /bak/staging-@today.sql
        $compare @dev-db :to @production-db then $save the @result :to /tmp/diff.sql
        $convert /tmp/diff.sql :to #sql then $execute :on the @production-db
        if any @errors ?occurred
                $restore the @production-db :using /bak/staging-@today.sql then
                $email @errors :to admin@acme.com

;; routine leveraging of web-based apis
$monitor the @arrival-time of #flight AC207 :for any ?changes
if @result ?changes $text @it :to @my-cell

;; custom scripting opportunties
every 5 minutes :for the next 5 days
    $search http://vancouver.craigslist.com :for "radiohead" or "radiohead tickets"
    if a @result is ?found $extract any #email-addresses or #phone-numbers then
    $text @them :to 778-319-4280
```

7. Internal Design

TYP is designed around the guiding precepts of self-containment and consistency. Self-containment describes a property of programming languages whereby they are defined largely in terms of their own constructs. To put it another way, TYP is written almost entirely in TYP. The core concept of TYP is types, i.e., *#database* or *#client*. These types, and a few core language features such as variable binding and control flow are written in a LISP variant called Racket. Almost every other concept introduced in this paper is written in TYP itself. For example, actions, prefixed with $, i.e., *$send* or *$delete*, are really just types under the hood. This is more readily apparent if you look at the code to define a new action.

```
$define #action my-action ...
```

Actions are not the only types in TYP. The remaining primary concepts such as properties, states, parameters and events are also defined as types, or as aliases of existing expressions. As outlined in section 5.8, aliasing provides a powerful means by which TYP has been built in itself.

Self-containment leads naturally to consistency. When TYP introduces a semantic form, it should hold universally throughout the language, rather than in restricted contexts or scenarios. This allows programmers to intuitively build upon previous understanding as more advanced concepts are digested. For example, recall that variables *and* properties are both prefixed with @, as in, *#client @first-name*, or *$subtract @costs :from @revenues.* At first glance this may appear redundant, but in fact there are no variables in TYP. Much as actions are simply types with convenient shorthands built by pattern matching, variables are actually *properties* of the *#environment* type. Thanks to the default types and actions introduced in section 5, most programmers needn't concern themselves with these sorts of details.

```
$define @my-var 3 ;; is translated by TYP into
$add #property my-var :to #environment then $set @it :to 3
```

Similarly, where you've seen "the @result", *@result* is actually a property of the previous *#expression*.

8. Conclusion

TYP aims to improve the efficiency with which human beings read and understand source code. This has traditionally led to major increases in developer productivity, allowing only a few programmers to accomplish what previously would have taken several dozen people or more; indeed, TYP already provides a measurable edge over contemporary languages for the developers that use it to deliver custom software to their clients. This domain is the first of many for which TYP could be of value, since TYP is basically an elegant wrapper around more complex functionality. Future versions of the language may allow the underlying functionality to be provided in a variety of languages (TYP already interoperates with Racket, Scheme and C, with planned incorporation of Javascript in the future). Ostensibly, TYP could become a protocol for various languages which would enable a wide range of programming tasks to be constructed by the end user without the technical expertise of a programmer.

In a future that promises everyday objects will be connected to each other producing ever increasing masses of data, the opportunities for predictive insights and automation potential is tremendous; but because of an ever increasing shortage of talented developers it is imperative that end users be empowered to leverage these opportunities in custom ways. In much the same way that Microsoft Excel enabled a legion of amateur financial analysts, TYP could unleash the creative power of motivated end users that want to build and automate existing systems with relative ease, and without reliance on professional computer programmers.