The Monad.Reader Issue 9: Summer of Code Special

by Mathieu Boespflug ⟨mboes@tweag.net⟩
and Jason Dagit ⟨dagit@codersbase.com⟩
and Thomas Schilling ⟨nominolo@gmail.com⟩

November 19, 2007

Wouter Swierstra, editor.

Contents

Editorial

by Wouter Swierstra ⟨wss@cs.nott.ac.uk⟩

For this issue of The Monad.Reader, I asked all the student participants of Google's Summer of Code mentored by Haskell.org to write something about their project. I was very pleased indeed when three of the nine students agreed to write an article for this issue: Thomas Schilling describes the Haskell Cabal and his work on Cabal Configurations; Jason Dagit gives an introduction to darcs's theory of patches; Mathieu Boespflug presents his work on the type checker behind nhc98 and Yhc. These three articles together find a rather nice balance between practical tools and theoretical challenges.

As an editor, it is my responsibility to say a few words about the other Summer of Code projects. To make the task of writing the editorial somewhat less tedious, I chose to write a short haiku about all the remaining projects. This proved quite a challenge! Hopefully my (lack of) poetic talent should convince next year's students to write an article, rather than be subjected to this public humiliation:

Abstracting Parsec
over any input type.
The world on a String.

To load code faster
and save disk space, GHC
must share libraries.

Improving Hackage.
Daemons can build your package
and Haddock your code.

Binding to libcurl.
Get your data from the web.
Who needs Firefox?

Haskell IDE,
will it ever beat Emacs?
Only time will tell.

A tracer that can't
cope with Cabalized code is
an old Hat, indeed.

Why haikus you ask? Well: Why should I squander, a boatload of syllables? Seventeen will do.

Cabal Configurations

by Thomas Schilling ⟨nominolo@gmail.com⟩

The Haskell Cabal is a tool specifically designed to help you package and release your Haskell code. Additionally, it provides a simple interface to common build and post-processing tasks such as running a test suite or generating Haddock documentation. Cabal configurations are a major new feature of the latest release.

Introduction

The Haskell Cabal [1] aims to provide a build environment suitable for most Haskell libraries and executable programs. Cabal itself is a Haskell library and comes with a few front-ends that utilize this library to manage Haskell packages. At the moment, only command-line based tools exist, but graphical front-ends, perhaps integrated into an IDE, are possible in principle. A version of Cabal is shipped with each GHC [2] and Hugs [3] release, but it is meant to be usable with any Haskell compiler or interpreter. Cabal is part of a larger infrastructure for distributing and organizing Haskell libraries and programs. Most Cabal packages are available from the Hackage [4] website.

If you want to install a Cabal package, you just need to download the sources and execute the following three commands:

```
runhaskell Setup.lhs configure --prefix=$HOME --user
runhaskell Setup.lhs build
runhaskell Setup.lhs install
```

This will install and register the package for the current user only, and therefore does not require administrator privileges. Cabal provides a multitude of options; for more information please consult the user's guide [5].

Using `cabal-install` [6] this process is even simpler. The `cabal-install` tool can fetch a package and all its dependencies from Hackage and install everything with a single command.

For example, to install *xmonad* version 0.4, you would just need to execute the following command:

```
cabal install xmonad-0.4
```

Developers can use Cabal to avoid writing complex Makefiles, and subsequently distribute their packages via Hackage. For packages with complex requirements Cabal allows different build types, such as the *autoconf* [7] tools. Additionally, developers can add custom build steps by means of a hooks interface[1].

The remainder of this article will describe the basic workings of Cabal, describe the new configurations feature, and will give a brief overview of which issues are currently being worked on.

Package Descriptions

So what do you need to do to release some Haskell code as a Cabal package? For most libraries and applications, this requires two simple steps:

1. create a package description file;
2. and create a *Setup.lhs* (or *Setup.hs*) file.

Most *Setup.lhs* files are just the trivial program shown in Figure 2.

The package description is a file named *packagename*`.cabal` and contains meta-data as well as a technical description of the package. Meta-data might consist of things like the package name and version, a textual description of the package, etc. The technical description of the package comprises things such as exposed interface, packages and tools required to build this package, command line options to the compiler, etc. A sample description is shown in Figure 1 (see below for a further description.)

Building a Cabal package is done by invoking the setup script with the desired command – the simple three-step install process we described in the introduction is one example of using Cabal. These commands will call into the Cabal library and invoke the necessary system commands. You can also use similar commands to bundle your source code, generate Haddock documentation, or run a test suite.

In Cabal 1.2.0 the syntax changed slightly, but is backwards-compatible with older package description files. The example in Figure 1 shows the typical structure of a package description file. It consists of a global part, that contains general information that applies to the package as a whole, followed by possible flag definitions, an optional library section, and an arbitrary number of executable sections.

The package described in Figure 1 contains one library (named like the package) and one executable named *ex1*. The library exports the three modules named

[1] Unfortunately, there is no reliable hooks interface at the moment, since parts of the Cabal API are occasionally being changed to incorporate new requirements.

```
package:       Foo
version:       0.42
cabal-version: >= 1.2
license:       BSD
license-file:  COPYING
copyright:     Jane Doe <jane@example.com>
category:      Example
synopsis:      An example package for TMR.
description:   This is just an example package.
               .
               We can have newlines in any field value.

flag UseTypeMagic
  description: Implement most features in the type-system.
               Requires a sufficiently smart compiler.
  default: False

library
  exposed-modules: Example1, Example2, Example3
  build-depends:   base >= 3.0 && < 4.0, filepath > 1, mtl >= 1.0
  if os(windows)
    build-depends: Win32
  else
    build-depends: unix
  if flag(UseTypeMagic)
    extensions:      MultiParamTypeClasses, FunctionalDependencies
    exposed-modules: Example4

executable ex1
  main-is: Main.hs
  build-depends: regex
  other-modules: Example1
```

Figure 1: An example package description file.

```
#!/usr/bin/runhaskell
> module Main where
> import Distribution.Simple
> main :: IO ()
> main = defaultMain
```

Figure 2: The default `Setup.lhs` file.

Example1 through *Example3*, and, optionally, *Example4* if the package user wishes to build it and the compiler provides the necessary features. The executable has the main file *Main.hs* and needs the *Example1* module from the library. To build the library, the user must have the packages *base*, *filepath*, *mtl*, and, depending on the operating system, either *Win32* or *unix* installed. To build the executable, the *regex* package has to be present as well. Each package may (and should) be constrained by a version range.

The ability to specify different variants, or **configurations**, of a package was the contribution of this Google Summer of Code project.

Cabal Configurations

Allowing different package descriptions depending on the system, and even enabling or disabling features as a user sees fit, has been a highly desirable feature – but no one seemed to be willing to implement it, even after the basic design had been agreed on [8]. Many package authors used hooks in setup scripts, but this led to needless duplication of code and was not transparent to tools that read the package description. Furthermore, many packages could not be built with multiple versions of GHC because the *base* package was split up into many smaller packages, thus essentially every package had to do some kind of dispatch on the version of the *base* package it is being compiled with.

Cabal configurations enable conditional parts in package descriptions. In order to check if a package can be built on a given platform or to figure out which other packages it depends on, it is now necessary to evaluate any conditions and to merge the field descriptions of all the selected branches. For most fields, merging means appending the fields from inside the conditional to the parts outside the conditionals, e.g.

```
build-depends: foo
if c
  build-depends: bar
```

will be merged to

```
build-depends: foo, bar
```

if c evaluates to *true*. Boolean fields are combined using conjunction. Some fields cannot be merged, resulting in an error during the configure phase.

Conditions are formed with predicates combined using the usual boolean operators `&&`, `||`, and `!` (negation). Predicates as of Cabal 1.2.0 are:[2]

`os(...)` tests on the operating system the package is being compiled for. For example `os(windows)`.

`arch(...)` tests on the architecture. For example `arch(ppc)`.

`impl(...)` tests on the implementation type and version. For example, `impl( GHC >= 6.8 )`.

`flag(...)` returns the current value of the specified flag, see below.

Tests on system parameters are rather simple. However, packages sometimes come with features, not every user wants to use (for example debugging support) or can use different packages to implement the same features (e.g., use different GUI toolkits), depending on what the user requests. To enable leaving those choices to the user Cabal allows definitions of **flags**. A flag has a name, an optional description, and a default value. Unless specified differently, a flag defaults to *True*. The idea behind this is, that typically flags represent features, and most users want to use all features unless there is a good reason not to (e.g., the compiler does not provide certain features.)

Flag Resolution

A flag can be assigned manually, during the configure phase via the `--flags` or `-f` option. To force a flag to *True* simply list the flag name as the argument. To force a flag to *False* list the flag name with a `"-"` in front of it. You can also specify multiple flags at once, for example

```
runhaskell Setup.lhs configure --flags="gtk -debug"
```

A forced flag will always be assigned the value as specified by the user, thus wherever it occurs in a condition, it will always evaluate to that value.

The tricky case is when the user does not specify an explicit value for flags. One choice would be to always assign the default value, and let the user specify

[2] A possible fifth predicate `package(...)` is being worked on that allows adding different dependencies, based on the version picked for one package. See the original proposal [9]

a custom value if the package fails to configure due to missing dependencies. The user would then have to figure out which flags caused it to fail and whether there is an assignment that allows the package to build successfully. To make things easier, Cabal takes a different approach. All flags not explicitly set by the user are considered variable and Cabal tries to find a suitable assignment, that is compatible with the packages available on the system. The algorithm is very simple, and can probably be best described by the following code snippet

```
findAssignments :: [Flag] → Maybe [(Flag, Bool)]
findAssignments varFlags =
  find satisfiesDependencies assignments
  where
    assignments :: [[(Flag, Bool)]]
    assignments = mapM (λf → [(f, flagDefault f)
                             , (f, ¬ (flagDefault f))])
                       varFlags
```

Note that *mapM* here works in the list monad. That is, given three flags a, b, and c, each defaulting to *True*, the above code will try all possible flag assignments in the order

- a=*True*, b=*True*, c=*True*
- a=*True*, b=*True*, c=*False*
- a=*True*, b=*False*, c=*True*
- a=*True*, b=*False*, c=*False*
- ...

This algorithm does of course have exponential complexity in the number of variable flags. At the moment, Cabal does not do any special optimizations, since the number of flags is usually sufficiently low that this shouldn't be an issue. (Having a SAT-solver in Cabal is also not desirable.) Note also, that the flags are tried in the order in which they are listed in the package description file. It is therefore possible to help Cabal a little by listing the flags that are most likely to cause problems last, so that an alternative for those flags is tried first.

Future Work

Even though Cabal configurations solved an important issue, there is still much, much work to do. Package management is a complex and loosely-defined problem with many challenges and many design decisions that need to be made. Here are a few of the problems the Cabal developers are still working on:

Cabal Frontends. The aforementioned `cabal-install` is available on Hackage and makes installing packages from Hackage very easy. A graphical tool, that is

more friendly towards newcomers (or Windows users) would be very useful.

Package Dependency Testing. At the moment, most packages describe dependencies in a very simple way. They usually just list the package name or a minimum version of the package. To make sure that packages are built correctly and work as expected, however, it is necessary that Cabal has a reliable way to pick the right dependencies. A versioning policy can help describe when a new release of package might possibly break dependent packages, but it might also inhibit the adoption of new versions since dependencies must first "bless" the new version of their dependency. The final solution must be a trade-off between several such issues. The discussion is ongoing [10, 11].

Module Dependency Tracking. At the moment, Cabal relies on GHC's `--make` feature to build all dependent modules. This works well enough, but is not portable and doesn't deal well with things like invoking pre-processors. Many computers today are multi-core, so we'd also like to take advantage of this to speed up compilaton of packages. To solve all those issues, Cabal will eventually contain a rule-driven engine, similar to the *make* tool, and will be able to track module dependencies and compile all modules individually and, wherever possible, in parallel. Unfortunately, the details are very tricky and hard to get right.[3]

Some packages with complex build requirements, e.g., *gtk2hs* [13], still cannot be built using Cabal. Ultimately, Cabal tries to support building the vast majority, if not all, of the Haskell packages people write.

Summary

Cabal is a useful tool for every Haskell hacker and is being improved continuously. Designing a package tool that aims to be useful to so many people is hard, but hacking on it is rewarding – you will definitely get user feedback!

Thanks

I'd like to thank Björn Bringert for pointing me to this Summer of Code project suggestion, Isaac Potoczny-Jones for being my mentor, and most of all Duncan Coutts for being very responsive about discussions and co-hacking on Cabal. Many thanks also to all those who sent feedback through the mailing lists and IRC channels.

[3] A recent mailing list message describes the current status [12].

About the author

Thomas Schilling has a B.E.Sc. in Computer Science from Dresden University of Technology in Dresden, Germany. He is now writing his Master's thesis at Chalmers University of Technology in Göteborg, Sweden. He hopes to get his M.Sc. at the beginning of 2008 and will try to get some kind of employment as a Haskell hacker after that.

References

[1] `http://www.haskell.org/cabal`.

[2] `http://www.haskell.org/ghc/`.

[3] `http://www.haskell.org/hugs/`.

[4] `http://hackage.haskell.org`.

[5] `http://www.haskell.org/ghc/docs/latest/html/Cabal/`.

[6] `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/cabal-install`.

[7] `http://www.gnu.org/software/autoconf/`.

[8] `http://www.mail-archive.com/cabal-devel@haskell.org/msg00282.html`.

[9] `http://www.haskell.org/pipermail/cabal-devel/2007-October/001189.html`.

[10] `http://hackage.haskell.org/trac/ghc/wiki/PackageCompatibility`.

[11] `http://haskell.org/haskellwiki/Package_versioning_policy`.

[12] `http://www.haskell.org/pipermail/cabal-devel/2007-October/001297.html`.

[13] `http://haskell.org/gtk2hs`.

Darcs Patch Theory

by Jason Dagit ⟨dagit@codersbase.com⟩

Darcs is a free, open source source code revision control system implemented in Haskell. One of its distinguishing features is the theory of ***patches****, that provides the theoretical foundations on which the tool is built. In this article, I will try introduces the basic ideas and operations of patch theory. We setup the conflict problem and we hope to explain the solution in later installments of* The Monad.Reader*.*

Basic Elements of Patch Theory

The goal of patch theory is to describe changes to a repository. Although patch theory could be treated purely as a mathematical abstraction, we focus on the details that relate to Darcs [1]. Please pay close attention to the relationship between **changes** and **patches.**

Definition 1. A **repository** is a set of changes.

Here we specifically mean a mathematical set; we mean to emphasize that the order of the elements is unimportant. This implies that merging two repositories should simply be the union of those sets.

Definition 2. In the abstract sense, a **change** is just an element of a repository. In Darcs, each change modifies the working copy or other changes.

Two changes are the same if they have the same effect, but it is important to notice that we cannot compare two changes unless they modify the same thing, which is to say they originate from identical states.

Definition 3. A **context** is the state after some number of changes have been added.

In order to manipulate changes, Darcs stores each change using a patch.

Definition 4. A **patch** is a fixed and concrete representation of a change. All discussions of patches in this document assume the representation used by Darcs, unless noted otherwise.

Each patch has two contexts associated with it. There is the context that the patch originates from and the context resulting from the patch. Given a set of patches we can only form sequences from the patches if their contexts match up. Additionally, we can only apply the changes represented by a patch if we have our data in the context required by the patch. Initially, these two requirements seem to go against the definition of a repository.

We want our repositories to store unordered sets of changes, but we have just stated that given a set of patches (which represent our changes) that there is an implicit order we must adhere to in order to apply the patches.

The Famous Commute

One way to get around the implicit ordering constraint described above is to create new representations of each change (i.e., new patches) so that they are in a different order.

Before we explain commute, here is an introduction to our notation. In the following example we use capital letters to represent patches and lower cased super scripts are used to mark the contexts.

To denote the patch A from context o to a, we would write ${}^{o}A^{a}$. A repository might contain two patches, ${}^{o}A^{a}$ and ${}^{a}B^{b}$, in which case we could put them in a sequence and simply write, ${}^{o}A^{a}B^{b}$. Note that since the ending context of A matches the starting context of B we only write the 'a' once. Often when the contexts are understood, they are omitted and just the patches are named.

Suppose we had two changes, C_1 and C_2 such that ${}^{o}A_1^{a}$ and ${}^{e}A_2^{c}$ both represent C_1 and, similarly, both ${}^{o}B_1^{e}$ and ${}^{a}B_2^{d}$ represent C_2. We could then write two different sequences of patches for our two changes. We could write,

$$ {}^{o}A_1^{a}B_2^{d} $$

or,

$$ {}^{o}B_1^{e}A_2^{c}. $$

Since both sequences represent having both changes their final contexts are actually the same and so we have

$$ c = d. $$

Patch theory defines a swap operation between two patches with suitable contexts called commute. The purpose of commute is to take two patches, which can

be placed in sequence, and create different representations of each patch so that the sequence order is reversed. Using the example above, it is possible to switch between the two sequences by commuting the two patches.

Not all patches can have their order reversed by the commute operation. For example, suppose one change creates a file and another change modifies the contents of that file. Since you cannot modify the contents of a file that hasn't been created, all patches representing modifications to the file must come after the patch that creates the file. When commute fails for this reason, we say that one patch depends on the other.

There are also several properties that each patch type must satisfy, but not all are listed here. For example, we must be able to invert each patch, or undo it's change. The inverse of patch B is denoted, $\overline{B}$.

Additionally, there are three properties which combined give an elegant way to merge two patches, but these properties are beyond the scope of this article. Instead of listing the properties, we give a simple example of how merge works.

Suppose we have the patches

$$^{o}A^{a} \text{ and } ^{o}B^{b}$$

and we would like to merge them. This requires us to find patch representations which can be put in sequence. Suppose we apply the inverse of A to get

$$^{o}A^{a}\overline{A}^{o}.$$

By the (unlisted) properties of the commute function we know that when commute succeeds we can commute

$$^{a}\overline{A}^{o}B^{b}$$

to get

$$^{a}B_1^{c}\overline{A}_1^{b}.$$

So now we can write the following sequence of patches

$$^{o}A^{a}B_1^{c}\overline{A}_1^{b}.$$

We can always remove patches from the end of a sequence by just throwing it away. We can remove patch from the right end to get the merged sequence,

$$^{o}A^{a}B_1^{c}.$$

The reader is encourage to check the Wikibook [2] for a clear, through and easy-to-read explanation of the merge algorithm. The reader should also verify that the patches can be merged in the other order, e.g. starting with patch B.

The Infamous Conflict

In the previous section, during the merge we assumed that the commutes always succeeded. This is not always the case. When we are performing a merge and two patches fail to commute, we say that the patches conflict with each other. This could happen if the two patches try to modify the same line of a file. So far we have two different times when commute can fail and both times means something different: in the context of a merge, it means the patches are conflicting; but when the patches are already in a sequence and fail to commute, it means there is a dependency between the patches. This difference is subtle but significant.

The conflict problem is how to retain the property that repositories are an unordered set of changes, but merging these sets can fail to produce a set of patches that can be ordered sequentially (i.e., merged).

There seems to be two main ways of solving the problem. One way is to hold all of the patches in a sequence using special patches to note the places where the sequence forks (current darcs does this with mergers and conflictors but the design is flawed). The other approach is to store all the patches in the repository and to "disable" some patches until the conflicts go away. The latter approach is easier conceptually, but requires recomputing the sequence of patches more frequently. For the Summer of Code we chose to implement the latter approach.

The conflict problem is worsened by scenarios where the user wants a patch that depends on a "disabled" patch. This requires Darcs to store all patches it has seen and have a way to determine when a new patch represents a change that has already been "disabled" (possibly by different patch representing the same change). This requirement led us to implement a "remerge" algorithm which is capable of taking a sequence of disabled patches and enabling as many of the patches as possible. This way it is possible to merge two repositories by disabling any possibly conflicting patches and then re-enabling some of them. We also added new patch types to allow the user to specify which changes should be active and which ones should be disabled.

References

[1] `http://darcs.net/unstable`.

[2] Wikibooks. Understanding darcs/patch theory. `http://en.wikibooks.org/wiki/Understanding_darcs/Patch_theory`.

TaiChi: how to check your types with serenity

by Mathieu Boespflug ⟨mboes@tweag.net⟩

The summer of 2007 was to be about hacking nhc98 and Yhc, two Haskell 98 compilers with common ancestry and some common code. My project focused on the front-end and, in particular, the type checker. In any Haskell compiler, the typechecker must infer the types of all expressions and subexpressions in a program with respect to the primitives of Haskell and other declarations. This information is useful for the later phases of the compiler, but more importantly, it enforces the static semantics of Haskell: rejecting any ill-typed program and hopefully giving helpful information to the user about why the program is ill-typed and how to fix it.

Unfortunately, the typechecker for nhc98 and Yhc hasn't kept up with the pace of the development of other Haskell compilers, and sometimes works in a rather obscure or unusual manner. The error messages can be quite unhelpful; sometimes the typechecker may even go wrong!

Enter TaiChi: a framework for writing typecheckers. The goal is to reduce the implementation of typechecking to a small, simple and clearly defined set of pluggable components. We try to make experimentation with new type systems as simple as possible and hopefully make the maintenance of a compiler a bit easier. We aim to provide good type error messages. Efficiency is not a primary concern.

The underpinning of TaiChi is the HM(X) theoretical framework set forward by Sulzmann et al. HM(X) is a general framework for typechecking Hindley-Milner style type systems. It describes type checking as a constraint satisfaction problem. It is general enough to handle numerous variations and extensions of the polymorphic λ-calculus. In particular, it is possible to specify the semantics of Haskell's type classes in terms of constraint handling rules (CHR), the usage of which we'll present later in this article. The main benefit of this approach is to support uniform construction of type inference algorithms, easing maintenance and experimentation with new type system extensions.

But first let's take a closer look to the formalism that lies at the heart of the framework: constraint logic programming. We will then give a feel of how to write a basic typechecker using TaiChi for your favourite ML-like or Haskell-like language (your **target language**) and we'll try to give an idea of TaiChi's current status and where it is heading towards.

Constraint logic programming

We view typechecking as an exercise in logic programming. The first ingredient in a logic program is the term language to manipulate. We take a term to be either a variable or an application of a **function** to a number of terms. The *CTerm* data type below describes the types that we will work with.

```
type Name = String
data CTerm = TVar Name
   | TApp Function [CTerm]
```

The *TVar* constructor corresponds to type variables; the *TApp* application corresponds to the application of a type constructor to its arguments.

Functions can be built using the *Func* constructors Func, that we have omitted. The *Func* constructor takes a name and an arity. Here are two example *CTerm* combinators that we will use to write the types of functions and tuples respectively:

```
x -:> y = TApp (Func "->" 2) [x, y]
triple x y z = TApp (Func "Tuple3" 3) [x, y, z]
```

In essence, a function is the same as Prolog structures. For purposes of this presentation terms are first-order, just like in Prolog. Now we can think of a constraint logic program as a set of clauses relating variables in a term, constraining the possible values those variables can be instantiated with. We define a constraint inductively as an application of a predicate to some terms or a conjunction of two constraints:

```
data Constraint = CTrue
   | CApp Predicate [CTerm]
   | CConj Constraint Constraint
   | CExist [CTerm] Constraint   -- Invariant: terms are unique vars.
```

Note that variables can be existentially quantified in constraints. The *Predicate* type has a *Pred* constructor, very similar to the *Func* constructor we have seen above.

When we collect constraints on types, for instance, when we carry out Hindley-Milner type inference, we'll usually only need the equality predicate:

$$a \doteq b = \mathit{CApp}\ (\mathit{Pred}\ \texttt{"="}\ 2)\ [a, b]$$

Of course, we are free to add more constraints, using the *Pred* constructor.

We combine constraints to form a larger constraint, as in the following (where *Bool* and [] are constants):

$$\begin{aligned} & \mathit{t3} \dot{\to} \mathit{t3} \doteq \mathit{Bool} \to \mathit{t2} \\ \wedge\ & \mathit{t5} \dot{\to} \mathit{t5} \doteq \mathit{Bool} \dot{\to} \mathit{t4} \\ \wedge\ & g\ (\mathit{t0}, \mathit{t1}) \\ \wedge\ & \mathit{t1} \doteq [\,] \\ \wedge\ & \mathit{tuple2}\ (\mathit{t2}, \mathit{t4}) \end{aligned}$$

With a representation of constraints in our toolbox, we'd like a way to reason about these constraints. For one, we would like to simplify and possibly solve the constraints. For this it is convenient to wrap up a constraint into a package, with a name and a set of variables that can be instantiated with arbitrary terms to form a new constraint. This is expressed by a **rule**, with a **head** and **body** as constituent parts, for reasons that will become clear.

$$\begin{aligned} & \textbf{data}\ \mathit{Rule} = \mathit{Head} :\!- \mathit{Goal} \\ & \textbf{data}\ \mathit{Head} = \mathit{Head}\ \mathit{Predicate}\ [\mathit{CTerm}] \\ & \textbf{data}\ \mathit{Goal} = \mathit{GAtom}\ \mathit{Predicate}\ [\mathit{CTerm}] \\ & \quad \mid \mathit{GCons}\ \mathit{Constraint} \\ & \quad \mid \mathit{GConj}\ \mathit{Goal}\ \mathit{Goal} \end{aligned}$$

You can recognize the structure of a rule as similar to the definition of a Prolog predicate. A rule does indeed fulfill a similar role. Note that the body of a rule is of type *Goal*, rather than a constraint. This is because a rule can, in addition to holding a constraint, 'call' another rule, with the same semantics as in Prolog. Therefore, we define a logic program as follows:

$$\textbf{type}\ \mathit{Program} = [\mathit{Rule}]$$

Given a logic program (i.e., a set of rules) $\mathcal{P}$, one may ask to solve a given goal, or in Prolog terms, perform a query.

$$\mathit{solve} :: \mathit{Program} \to \mathit{Goal} \to \mathit{Answer}$$

To solve a CLP goal is to recursively substitute any instance of the head of a rule in the goal with the body of the rule, renaming free variables if necessary to

avoid name clashes, repeating the process until all that is left is a conjunction of constraints. More precisely, the goal is rewritten by SLD-resolution with respect to $\mathcal{P}$, just as in Prolog.

An example derivation would be as follows. Given the program

$$\begin{array}{lll} f(t_1, t_2) & :- & t_1 \doteq \texttt{Bool} \wedge t_2 \doteq \texttt{Int} \\ g(u_1, u_2) & :- & u_1 \doteq u_2 \wedge f(\texttt{Bool}, u_2) \wedge u_1 \doteq u_2 \dot{\rightarrow} u_2 \end{array}$$

rewriting the goal $g(a, \texttt{Int})$ where a is a variable yields

$$\begin{array}{rcl} g(a, \texttt{Int}) & \rightsquigarrow & u_1 \doteq a \wedge u_2 \doteq \texttt{Int} \wedge u_1 \doteq u_2 \wedge f(\texttt{Bool}, u_2) \wedge u_1 \doteq \texttt{Int} \\ & \rightsquigarrow & u_1 \doteq a \wedge u_2 \doteq \texttt{Int} \wedge u_1 \doteq u_2 \wedge t_1 \doteq \texttt{Bool} \wedge t_2 \doteq u_2 \wedge \\ & & t_1 \doteq \texttt{Bool} \wedge t_2 \doteq \texttt{Int} \wedge u_1 \doteq \texttt{Int} \end{array}$$

Of course this last set of constraints on a can be simplified to

$$a \doteq \texttt{Bool} \wedge a \doteq \texttt{Int}$$

This set of constraints cannot be satisfied.

Introducing the TC monad

Given facilities for constraint logic programming and basic functions on variables such as finding free variables, renaming and so forth, we can already perform basic typechecking. But before we demonstrate this in more detail, we will write some additional machinery to keep track of type environments, report errors to the user and generate fresh variable names. Enter the TC monad, a thin layer wrapped around most of the TaiChi library.

Our goal here is to define a base monad including some basic plumbing to help type checking, on top of which we can build additional mechanisms. It would be very convenient to use GHC's newtype deriving here, and get many definitions for free, but this feature is not available in all Haskell compilers. We layer monad transformers much like in the mtl library,[1] but the approach here is a bit more general.

The TC monad is defined as follows, using an assemblage of monad transformers. The idea is to make it trivial to add more transformers as necessary.

```
newtype TC v ty a = TC{ unTC :: StateT [v] (ReaderT (Env v ty)
  (Either String)) a }
```

Add a few more trivial instance definitions and we already have a monad from which we can generate fresh variables, set local type environments and abort due

[1] `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/mtl-1.0`

to an error. Fresh variables are modelled by seeding the state in the monad with an infinite list of names. Each time a new name is required, we pop the head off the list and update the state accordingly. This functionality is partially captured in the following class, of which TC is an instance:

```
class Monad m ⇒ MonadTC m v ty r | m → v, m → ty, m → r where
  environment :: m (Env v ty)
  setEnvironment :: Env v ty → m a → m a
  lookupVar :: Eq v ⇒ v → m (Maybe ty)
  withBinding :: Eq v ⇒ (v, ty) → m a → m a
  freshVar :: m v
  split :: m a → m ()
  errorTC :: TCError → m a
  ...
```

Note the r type variable in the head of the class definition. This is used to layer additional monad transformers on top of TC. Monad transformers offer a *lift* method to promote actions of a monad M to actions of a new monad composed of M and some monad transformer T. Additional calls to *lift* will take the action even higher up an arbitrary size stack of monad transformers. But *lift* has the limitation that it does not cope with 'higher-order' monadic actions nicely, by which we mean actions that take actions as arguments, such as *withBinding* above.

Given instances of all methods of *MonadTC*, given in

```
instance MonadTC (TC v ty) v ty () where
  ...
```

we would like to get instances of these methods for any other monad that we layer on top of TC for free, with the guarantee that these methods will have the same behaviour in the layered monad as they do in the TC monad. Given a function *liftTC* :: $TC\ v\ ty\ a \rightarrow m\ a$ this is trivial for most methods above: by default define them in terms of *liftTC*. But how can we do this for methods that take monadic actions as arguments? We need some way of peeling off the layered monad transformer to obtain the TC monad underneath, modify it, then wrap it back up with the monad transformer. This is where *dropTC* :: $m\ a \rightarrow r \rightarrow TC\ v\ ty\ a$ comes in, the inverse of *liftTC* in some sense.

Notice the r argument. This is because, in a purely functional language, in general monads carry around implicit state. We could term this **input state**, for monads that allow actions to inherit state but never produce any, and **output state** for the converse. The *Reader* monad is an example of a monad with input state and the *Writer* monad an example of a monad with output state. The *State* monad has both input and output state. So to **destruct** the outer layer, we'll need

in general to provide some input state, such as the current environment to run the Reader monad. To **reconstruct** the outer layer such as it was before, we can simply use *liftTC*, but whose signature we need to modify slightly. We want *liftTC* as an inverse to *dropTC*, so we add the two methods to *MonadTC* with the following signatures:

```
instance MonadTC m v ty r | m → v, m → ty, m → r where
  liftTC :: (r → TC v ty a) → m a
  dropTC :: m a → (r → TC v ty a)
  ...
```

The redundant parentheses in *dropTC*'s signature are just to underline the symmetry between the two methods. All other methods have the same signatures as previously. As an example, a default implementation of *withBinding* would be

```
withBinding :: Eq v ⇒ (v, ty) → m a → m a
withBinding b m = liftTC (λr → withBinding b (dropTC m r))
```

Of course this approach is not possible with arbitrary monads. It goes without saying that layering the *IO* monad on top of *TC* in this manner is not possible. This approach only works for monads that are in some sense 'runnable,' from which we can extract a value and some hidden state. Note also that *liftTC* and *dropTC* as defined above only handle input state, though they can readily be extended to handle output state as well, hence allowing one to add a *StateT* layer. For other monads, one can always give explicit definitions for the higher-order monadic actions that *MonadTC* defines, since the default definitions for these methods will not suffice. The ability to add new monadic layers easily will be very useful: it makes the creation of new monads cheap and safe.

Writing some type rules

With all the basic infrastructure in place, we can now move on to the core task of writing a new type system: defining the type rules. What's more, writing these rules is all it takes to get the basic functionality of a type checker for your target language. Let's illustrate this with λ_{HM}, a tiny λ-calculus with polymorphic types.

```
data Expr = EVar Name
    | EAbs Expr Expr
    | EApp Expr Expr
    | ELet Expr Expr Expr
    | ERec Expr Expr
    | ETuple [Expr]
```

Typechecking using CLP consists of at least three phases. First we need to work out the type of an expression, defined recursively on the structure of expressions in the target language. Such a definition is given formally by type rules.[2] During typechecking, we must also accumulate constraints relating the inferred types of all sub-expressions to each other. We can define how this is done for the target language by means of a relation $\Gamma, E, e \vdash_{CG} (G \mid T)$, where Γ is a set of assumptions on the types of λ-bound variables in the current environment and G, T are respectively a conjunction of constraints and the type of e. Finally, E is used to keep track of those variables that are let-bound in the current scope. Table 1 shows the formal syntax driven definition of this relation. This style of definition may look scary at first, but is rather straightforward once you understand the mathematical notation. In a minute, we will see how all these symbols can be translated to Haskell.

Applying the constraints generated by $\vdash_{CG}$ to a type variable standing for the type of the input expression, solving them, finding the most general unifier and appropriately quantifying any free variables yields the **principal type** (or most general type) of the expression.

We can think of the constraint generation relation $\vdash_{CG}$ as a simple function taking as input some environments and an expression. Of course it is much more convenient and less error prone to hide the environments in a monad and get the monad to handle fresh variable generation. In general, we let the user use any monad he wishes and provide the following interface to type rules of the object language.

```
class TypeRules e m | e → m where
  genConstraints :: e → m (CLP.Goal, CLP.CTerm)
  genCLPRules :: e → m CLP.Program
```

By way of example, to typecheck expressions of λ_{HM}, we'll make use of the *CRGen* monad, a simple extension of *TC*: add an extra environment for let-bound variables.

```
newtype CRGen v a = CRGen{ unCRGen :: ReaderT [v] (TC v CTerm) a }
instance MonadTC (CRGen v) v CTerm [v] where
  liftTC = CRGen ∘ ReaderT
  dropTC m = runReaderT (unCRGen m)
instance Monad (CRGen v) where
  return a = CRGen (return a)
  m >>= k = CRGen (unCRGen m >>= λx → unCRGen (k x))
```

[2] Type rules omitted here since they are the usual ones for the polymorphic λ-calculus. They can be found in the literature describing HM(X) [1]

$$\text{(CGVar-x)} \quad \frac{(x : t) \in \Gamma}{E, \Gamma, x \vdash_{CG} (True \mid t)}$$

$$\text{(CGVar-f)} \quad \frac{f \in E \qquad t, l \quad \text{fresh}}{E, [x_1 : t_1, \ldots, x_n : t_n], f \vdash_{CG} (f(t, l) \wedge l = [t_1, \ldots, t_n] \mid t)}$$

$$\text{(CGAbs)} \quad \frac{E, \Gamma +\!\!+ [x : t_1], e \vdash_{CG} (G \mid t_2) \qquad t_1 \quad \text{fresh}}{E, \Gamma, \lambda x.e \vdash_{CG} (G \mid t_1 \to t_2)}$$

$$\text{(CGApp)} \quad \frac{E, \Gamma, e_1 \vdash_{CG} (G_1 \mid t_1) \qquad E, \Gamma, e_2 \vdash_{CG} (G_2 \mid t_2) \qquad t \quad \text{fresh}}{E, \Gamma, e_1\ e_2 \vdash_{CG} (G_1 \wedge G_2 \wedge t_1 = t_2 \to t \mid t)}$$

$$\text{(CGLet)} \quad \frac{\begin{array}{c} E \cup \{f\}, \Gamma, e_2 \vdash_{CG} (G \mid t) \\ \Gamma = [x_1 : t_1, \ldots, x_n : t_n] \qquad a, l \quad \text{fresh} \end{array}}{E, \Gamma, \text{let } f = e_1 \text{ in } e_2 \vdash_{CG} (G \wedge f(a, l) \wedge l = [t_1, \ldots, t_n] \mid t)}$$

$$\text{(CGRec)} \quad \frac{E, \Gamma +\!\!+ [f : a], e \vdash_{CG} (G \mid t) \qquad a \quad \text{fresh}}{E, \Gamma, \text{rec } f \text{ in } e \vdash_{CG} (G \wedge a = t \mid t)}$$

Figure 1: Formal definition of constraint generation relation

We will need a few accessor functions useful in the definition of rules.

```
withLetBinding :: Eq v ⇒ v → CRGen v a → CRGen v a
withLamBinding :: Eq v ⇒ (v, CTerm) → CRGen v a → CRGen v a
lamEnvironment :: CRGen v (Env v CTerm)
setLamEnvironment :: Env v CTerm → CRGen v a → CRGen v a
isLetBound :: Eq v ⇒ v → CRGen v Bool
freshTyVar, freshEnvVar :: CRGen v CTerm
```

We can give a rather direct implementation of the rules defining the behaviour of the typechecker for λ_{HM} in Haskell. To do so, we must write the following instance definition.

```
instance TypeRules HM.Expr (CRGen HM.Name) where
  genConstraints = genC
  genCLPRules = genR
```

$$\text{(RGVar)} \qquad E, \Gamma, v \vdash_{RG} \emptyset$$

$$\text{(RGAbs)} \qquad \frac{E, \Gamma +\!\!+ [x : t], e \vdash_{RG} P \qquad t \quad \text{fresh}}{E, \Gamma, \lambda x.e \vdash_{RG} P}$$

$$\text{(RGApp)} \qquad \frac{E, \Gamma, e_1 \vdash_{RG} P_1 \qquad E, \Gamma, e_2 \vdash_{RG} P_2}{E, \Gamma, e_1\ e_2 \vdash_{RG} P_1 \cup P_2}$$

$$\text{(RGLet)} \qquad \frac{\begin{array}{c} E, \Gamma, e_1 \vdash_{CG} (G \mid t) \qquad \Gamma = [x_1 : t_1, \ldots, x_n : t_n] \qquad l, r \quad \text{fresh} \\ E, \Gamma, e_1 \vdash_{RG} P_1 \qquad E \cup \{f\}, \Gamma, e_2 \vdash_{RG} P_2 \\ P = P_1 \cup P_2 \cup \{f(t, l) :\!- G \wedge l = [t_1, \ldots, t_2]\} \end{array}}{E, \Gamma, \text{let } f = e_1 \text{ in } e_2 \vdash_{RG} P}$$

$$\text{(RGRec)} \qquad \frac{E, \Gamma, e \vdash_{RG} P}{E, \Gamma, \text{rec } f \text{ in } e \vdash_{RG} P}$$

Figure 2: Formal definition of CLP rule generation relation (naive but mostly correct version)

We must now define the *genC* and *genR* functions that generate the constraints and rules respectively. The most important parts of the the *genC* function can be found in Figure 4. The focus here is on the definitions for λ-abstractions, applications and let declarations – we have left out several of the less interesting cases. The full definition is in `TaiChi/HM/TypeRules.hs` in the TaiChi source repository. Notice how closely the implementation mirrors the type rules in Figure 1.

Now each let definition induces a CLP rule. Before we solve the constraints collected by *genConstraints*, it is necessary to collect all such rules in the input expression. The CLP rules of an expression are given by the relation $\vdash_{RG}$, formally defined in Figure 2.

The formal specification, and accordingly the implementation of *genR* of $\vdash_{RG}$, are both rather straightforward. The definition of *genR* is listed in Figure 4. The only tricky part is the case of the let-binding, where *genConstraints* is called on the right hand side of a let-definition to infer the type and associated constraints of the let-bound symbol. In all the other cases, it is reassuring to see that the implementation of *genR* closely mirrors the formal rules in Figure 2. We'll now consider a few examples, before discussing how to deal with type classes.

```
genC :: HM.Expr → CRGen HM.Name (CLP.Goal, CLP.CTerm)
genC (EAbs (EVar x) e) = do
  t1 ← freshTyVar
  (goal, t2) ← withLamBinding (x, t1) (genC e)
  return (goal, t1 -→ t2)
genC (EApp e1 e2) = do
  t ← freshTyVar
  (g1, t1) ← genC e1
  (g2, t2) ← genC e2
  return (conjunction [g1, g2, GCons (t1 ≐ t2 -→ t)], t)
genC (ELet (EVar f) src target) = do
  a ← freshTyVar
  l ← freshTyVar
  (goal, t) ← withLetBinding f (genC target)
  tys ← mkLamTypes
  let g = conjunction [goal
    , GAtom (Pred f 2) [a, l]
    , GCons (l ≐ tys)]
  return (g, t)
  ...
```

Figure 3: The definition of the *genC* function

```
genR :: Expr → CRGen HM.Name CLP.Program
genR (EVar x) = return CLP.nullProgram
genR (EAbs (EVar x) e) = do
  t ← freshTyVar
  prog ← withLamBinding (x, t) (genR e)
  return prog
genR (EApp e1 e2) = do
  p1 ← (genR e1)
  p2 ← (genR e2)
  return (p1 ++ p2)
genR (ELet (EVar f) src target) = do
  (goal, srcTy) ← genConstraints src
  tys ← mkLamTypes
  l ← freshTyVar
  ty ← freshTyVar
  p1 ← genR src
  p2 ← withLetBinding f (genR target)
  let prog = p1 ++ p2 ++ [(Head (Pred (identify f) 2) [ty, l])
     :-conjunction [GCons (ty ≐ srcTy)
       , goal
       , GCons (l ≐ tys)]]
  return prog
genR (ERec f e) = genR e
```

Figure 4: The definition of the *genR* function

A small λ_{HM} example

If you are in the shoes of a language designer trying to get a quick and dirty type system going with little fuss, writing the above constraint and CLP rule generation rules is all you need to do. A series of small building blocks are available in `TaiChi/TypeCheck/` which you can use for common tasks, such as inferring the type of an arbitrary expression, or typechecking a set of top-level declarations in a module. Look inside `TypeCheck/Expression.hs` to find *tcExpression*:

$$
\begin{aligned}
&\mathit{tcExpression} :: (\mathit{TypeRules}\ t\ (\mathit{CRGen}\ v), \mathit{Identify}\ v\ \mathit{Name}) \Rightarrow \\
&\quad \mathit{Env}\ v\ \mathit{TyScheme} \quad \text{-- The types of all primitives.} \\
&\quad \rightarrow \mathit{Env}\ v\ \mathit{CTerm} \quad \text{-- The types of all lambda-bound variables in scope.} \\
&\quad \rightarrow [v] \quad \text{-- The infinite supply of fresh variables.} \\
&\quad \rightarrow t \\
&\quad \rightarrow \mathit{Either}\ \mathit{TCError}\ \mathit{TyScheme}
\end{aligned}
$$

This function will work with any abstract syntax tree over which type rules have been defined and for which symbols in the language can be mapped to strings. It will indicate failure of typechecking, but with only the type rules to go by, the type error messages will be rather cryptic and unhelpful.

Let's have a look at a simple example. Let *test1* denote the expression

$$\mathit{test1} = \lambda y \rightarrow \mathbf{let}\ f\ x = (y, x)\ \mathbf{in}\ (f\ \mathit{True}, f\ y)$$

We can find the type of this expression assuming that *True* is of type *Bool*.

```
ghci> tcExpression [] [] freshVars test1
Right (TSContext [t0] t0 -> tuple2(tuple2(t0, Bool),
                                   tuple2(t0, t0)))
```

That is to say,

$$\texttt{test1} :: \texttt{forall}\ \alpha.\ \alpha \rightarrow ((\alpha, \texttt{Bool}), (\alpha, \alpha))$$

And when introducing a type error in the program:

```
ghci> tcExpression [] [] freshVars test2
Left (t3 = t4 , t3 = Bool , t4 = Bool -> t9)
```

where

$$\mathit{test2} = \lambda y \rightarrow \mathbf{let}\ f = \mathit{True}\ \mathbf{in}\ (f\ \mathit{True}, f\ y)$$

A *Right* value tells us that typechecking succeeded and gives the inferred type. A *Left* value means that on the contrary typechecking failed. In the latter case TaiChi makes a small attempt at giving a hint of where the type error lies by returning a **minimal unsatisfiable subset** of constraints, a set of conflicting constraints such that removing any one constraint makes the set non-conflicting. In future work this set will be used as a basis for giving more helpful error messages.

Typing with classes

So far the only constraints we have used are equality constraints. This was sufficient for a basic Hindley-Milner type system. We can add more constraint predicates, to express richer systems, but one issue is that it is necessary to define in Haskell the meaning of any new constraint predicate we introduce. When solving constraints, encountering an equality predicate means unify both sides. Other predicates will mean something else. A very convenient and powerful way of specifying the semantics of new constraint predicates is to employ constraint handling rules. CHR's typically have two forms: simplification rules and propagation rules.

$$\begin{array}{lrcl} \text{simplification} & r1 & \equiv & c_1, \ldots, c_n \iff d_1, \ldots, d_m \\ \text{propagation} & r2 & \equiv & c_1, \ldots, c_n \implies d_1, \ldots, d_m \end{array}$$

where $r1, r2$ are rule names, $c_1, \ldots, c_n$ are CHR constraints and $d_1, \ldots, d_m$ are CHR or CLP constraints.

A simplification rule is to be interpreted as follows. Given a bag of constraints $\mathcal{B}$, if there are constraints matching the patterns in the head of the rule then remove those constraints and replace them with the body of the rule (right hand side of $\iff$). A propagation rule is interpreted the same way, save for the fact that for such a rule matching constraints are not removed from $\mathcal{B}$.[3]

A set of constraint handling rules taken together form a CHR program, and this program tells us how to manipulate constraints to form new constraints and simplify them. It turns out that as far as typechecking is concerned Haskell type classes can be mapped to an equivalent CHR program. The idea is that for each occurrence of a method of some type class in some expression, such as $(\equiv)$ of class *Eq* or *show* of class *Show*, we must generate some additional constraints that we'll call class constraints (e.g. *Eq Int* or *Show a*). These constraints are witness to the usage of such methods and restrict the inferred type with additional conditions. For instance if the user defines $f\ x\ y = x \equiv y$ then f can only be applied to arguments of some type t if t is an instance of *Eq* t. In Haskell we indicate this extra condition by denoting the type of $f :: Eq\ t \Rightarrow t \to t \to Bool$. Now if we instantiate t to *Int*, for instance by giving an explicit signature for f in the program, the *Eq Int* context becomes redundant and we omit it, because and instance *Eq Int* is defined in the Prelude.

This inspires the following natural encoding of type classes as CHR's. On the left, we write the Haskell code; on the right, we write the corresponding constraint.

class $C \Rightarrow TC\ a$ **where** $m :: D \Rightarrow t$	$TC\ a \Longrightarrow C$
instance $E \Rightarrow TC\ t$	$TC\ t \Longleftrightarrow E$

[3] This is a very swift overview of CHR's. The interested reader is referred to the literature about HM(X) [2] for more details.

For the *Ord* class, this would give:

class $Eq\ a \Rightarrow Ord\ a$ **where**...	$Eq\ a \Longrightarrow True$
instance $Ord\ Int$ **where**...	$Ord\ a \Longrightarrow Eq\ a$
instance $Ord\ a \Rightarrow Ord\ [a]$ **where**...	$Ord\ [a] \Longleftrightarrow Ord\ a$
instance $Ord\ a \Rightarrow Ord\ Bool$ **where**...	$Ord\ Bool \Longleftrightarrow True$

Extra CLP rules are generated mirroring the signatures of class methods. In general a method m of class TC will induce a CLP rule of the form given below.

$$m :: D \Rightarrow \tau \quad \vdash_{RG} \quad m\ (t) :- t \doteq \tau \wedge D \wedge TC\ a$$

In the above examples we would therefore have

$(\equiv) :: a \rightarrow a \rightarrow Bool$	$\equiv (t) :- (t \doteq a \rightarrow a \rightarrow Bool) \wedge Eq\ a$
$compare :: a \rightarrow a \rightarrow Ordering$	$compare(t) :-$ $(t \doteq a \rightarrow a \rightarrow a \rightarrow Bool) \wedge Ord\ a$

For the interested reader, a more adequate treatment of the use of CHR's for type inference in the presence of type classes is given elsewhere [3]. In TaiChi, the presence of CHR's means that the solving of constraints must be interleaved with calls to the CHR solver to iteratively simplify and rewrite constraints until no longer possible. We must therefore employ the *tcExpressionCHR* function, the analog of *tcExpression* with extra logic to handle CHR's.

Conclusion

TaiChi is lightweight and lays the groundwork for writing a typechecker succinctly, but still allows a lot of flexibility regarding the kind of type system you wish to implement. The HM(X) theoretical framework is a very powerful one indeed, allowing the expression of a rich set of extensions to the traditional Hindley-Milner type system. We have only considered here the addition of CHR's to the system to model Haskell style type classes, but it is also possible with CHR's to model multi-parameter type classes, functional dependencies, existential types, GADTs and many other familiar type system extensions. Moreover, TaiChi sports some rudimentary notions of model theory, allowing one to vary the constraint domain and solver to use, for instance to create useless if funny type systems with built-in notions of arithmetic over reals.

Currently TaiChi only provides a very rudimentary framework. Type errors are not handled in a user friendly way yet and some algorithms are very naive. Constraint rules have been implemented for most of Haskell 98 and are currently being integrated into Yhc. I plan to write support for nice error messages, using an approach directly benefiting from the principled use of constraints exposed here. I

also plan to substantially reduce the amount of work required to write constraint rules for common type system and language features. Further improvements will focus on fully implementing type classes as they are found in GHC and further type extensions. This will enable Yhc, nhc98 and any other compiler that decides to make use of TaiChi, to make quickly support many type system features that can be found in some other compilers.

This article is a small account of how to program using TaiChi, a project in a still nascent stage. I hope that this overview has given you a feel for how to program using TaiChi and de-mystified typecheckers somewhat. Type systems are not black magic, and set in the right theoretical framework understanding them can be rather straightforward. If you are interested, TaiChi can sure do with an extra pair of hands! The darcs repository is available at:

`http://code.haskell.org/~mboes/taichi/`

Acknowledgments

Many thanks to Martin Sulzmann, Peter J. Stuckey and Jeremy Wazny, whose ideas about constraint-based and CHR-based typechecking lay the foundations for TaiChi, and to Malcolm Wallace, my mentor.

About the author

Mathieu Boespflug has an MMath from the University of York in Computer Science and Mathematics. He was first introduced to functional programming in his first year at York. The grounds of Scheme were nice, but he was eventually tempted in exploring out into the highlands, where he found Haskell. He is now in the territory of O'Caml and Coq, starting a PhD pertaining to the compilation of term rewriting at INRIA Polytechnique, near Paris, although he hopes that the good old days of Haskell hackery will return someday.

References

[1] M. Sulzmann and P.J. Stuckey. HM (X) type inference is CLP (X) solving. **Journal of Functional Programming**, pages 1–33 (2007).

[2] T. Fruehwirth. Theory and practice of constraint handling rules. **Journal of Logic Programming**, 37(1):pages 95–138 (1998).

[3] P. J. Stuckey, M. Sulzmann, and J. Wazny. Type processing by constraint reasoning. In **Proc. of APLAS'06**, volume 4279 of **LNCS**, pages 1–25. Springer-Verlag (2006).