Haskell on Visual Studio

Simon Marlow
Simon Peyton Jones
What are we doing?

- Using Visual Studio’s customisation facilities (VSIP) to create a Haskell development environment.
- Plugging GHC itself into Visual Studio to provide Haskell support.
  - A new front-end on GHC to provide the interfaces that VS consumes.
  - Abstracting GHC as a library in order to do this.
Hasn’t this been done already?

- Several years ago, Daan Leijen wrote the Babel module for Visual Studio.
  - Babel is a higher-level abstraction over VS’s internals, that allows new language services to be provided with little effort. Adding new language support using Babel is much easier than programming to the VSIP interfaces directly.
  - Babel provides support for syntax colouring, on-the-fly error detection, Intellisense™, given some minimal language-specific support code.
  - Daan implemented some of these features for Haskell, on top of Babel. His implementation was in C.
We’re going further

- Still using Babel, but
  - Implementing our plugin in Haskell, using H/Direct to talk to the Babel COM interface.
  - Using a complete Haskell front-end (i.e. GHC itself) to get better on-the-fly checking of Haskell code in the development environment: static/dynamic semantics checking as you type.
  - Aim: to make a fully-fledged VS component that is downloadable for free (will require VS itself, though).
Features

- Syntax colouring
- Commenting/uncommenting code
- Error indications as you type:
  - parse errors
  - static semantics (out of scope vars, etc.)
  - type errors
- Brace matching
- Ask for the types of identifiers in the editor
- Jump to documentation for library identifiers
- Jump to binding site for an identifier
- Automatic indexing of a module (by function, type, class).
- Indentation/formatting support
- build / run the whole program (replaces ‘make’).
- integrate with cross-language tools (?)

The Story so far 😊

- Class hierarchy browser
- Debugging support
- Refactoring
-- cycle ties a finite list into a circular one, or equivalently,
-- the infinite repetition of the original list. It is the identity
-- on infinite lists.

\[
\begin{align*}
cycle & \colon [a] \rightarrow [a] \\
cycle \ [] & \quad = \text{error "Prelude.List: empty list"} \\
cycle \ xs & \quad = \text{xs' where xs' = xs ++ xs'}
\end{align*}
\]

-- take n, applied to a list xs, returns the prefix of xs of length n,
-- or xs itself if n > length xs.  drop n xs returns the suffix of xs
-- after the first n elements, or [] if n > length xs.  splitAt n xs
-- is equivalent to (take n xs, drop n xs).

\[
\begin{align*}
take & \colon \mathbb{N} \rightarrow [a] \rightarrow [a] \\
take \ n \ _ & \quad = \ [] \\
take \ _ \ [] & \quad = \ [] \\
take \ n \ (x:xs) & \quad = \ x : \text{take (n-1) xs}
\end{align*}
\]

\[
\begin{align*}
drop & \colon \mathbb{N} \rightarrow [a] \rightarrow [a] \\
drop \ n \ xs & \quad = \ text{xs if n > 0} \\
drop \ _ \ [] & \quad = \ [] \\
drop \ n \ (\text{xs}) & \quad = \ text{drop (n-1) xs}
\end{align*}
\]

\[
\begin{align*}
\text{splitAt} & \colon \mathbb{N} \rightarrow [a] \rightarrow ([a],[a]) \\
\text{splitAt} \ n \ xs & \quad = \ (\text{take n xs, drop n xs})
\end{align*}
\]

-- takeWhile, applied to a predicate p and a list xs, returns the longest
-- prefix (possibly empty) of xs of elements that satisfy p.  dropWhile p xs
-- returns the remaining suffix.  span p xs is equivalent to
-- (takeWhile p xs, dropWhile p xs), while break p uses the negation of p.
Insert a '('

Parse error is indicated at 'where'

Error also listed in the Task List; clicking here jumps to the error
The obligatory block diagram

Visual Studio

Babel

VSIP COM interfaces

Babel COM interfaces

Haskell Service

Invokes parser for error detection

Invokes lexer for syntax colouring

C/C++

Haskell

GHC

Next...
Next steps…

- GHC needs to track source locations much more accurately: when reporting an out-of-scope identifier, we need to know the exact start/end position of the identifier in the source so we can highlight it in the editor.
  - First step: replace GHC’s lexer with one that retains full positional info on each token.
  - This entailed a slight detour to work on Alex version 2.0 😊
  - Now: GHC has an Alex lexer, which is faster and more correct than the original.
  - The next stage is to push the positional information through the rest of the compiler.
GHC as a library

- GHC currently has three front-ends:
  - GHCi
  - --make
  - one-shot command line compilation

![Diagram showing GHC, GHCi, One-shot, --make, Haskell execution platform API, and GHC as a library]
GHC as a library

- GHC currently has three front-ends:
  - GHCi
  - --make
  - one-shot command line compilation
- There are potentially more:
  - GUI front ends
  - The VS plugin
- What is the underlying API that these front ends consume? Currently in GHC it is the compilation manager interface, but
  - this API is not as clearly defined as it could be
  - we will certainly need more functionality for VS than it currently provides.
- c.f. our earlier design work on the Haskell Execution Platform. Should we resurrect this idea?
Finally…

- We hope to get a “technology preview” out before too long, with just basic functionality.
- Feature suggestions are welcome.
- Help is welcome too: you need Visual Studio, and the VSIP SDK which is available for free (if you’re prepared to accept the license 😊).