My summer holiday

Extracting Haskell programs

Fredrik Nordvall Forsberg

Realizability seminar 13.02.2013

Implementing realizability

Theorem (Soundness)

Let M be a derivation of A from assumptions $u_i : C_i$ $(i < n)$. Then we can derive $\mathrm{et}(M)\ \mathbf{r}\ A$ *from assumptions* $x_{u_i}\ \mathbf{r}\ C_i$.

- Implemented in the Minlog proof assistant.

```
(proof-to-extracted-term (current-proof))
```

- The extracted program is correct by construction.

- A proof of this fact can be automatically generated.

```
(proof-to-soundness-proof (current-proof))
```

Correct but slow?

- Last week, Andy showed us an extracted SAT solver.
- However, he said that it needed 37 minutes to decide if you can fit 6 pigeons in 5 holes (with n.c. quantifiers).
- Extracted programs are terms in Minlog's internal representation, evaluated via NbE in Scheme.
- Are slow programs the price we have to pay for verified correctness?

Correct but slow?

- Last week, Andy showed us an extracted SAT solver.
- However, he said that it needed 37 minutes to decide if you can fit 6 pigeons in 5 holes (with n.c. quantifiers).
- Extracted programs are terms in Minlog's internal representation, evaluated via NbE in Scheme.
- Are slow programs the price we have to pay for verified correctness?
- No! I will show you how to reduce Andy's time to 0.340 s, without changing the program.

Correct but slow?

- Last week, Andy showed us an extracted SAT solver.
- However, he said that it needed 37 minutes to decide if you can fit 6 pigeons in 5 holes (with n.c. quantifiers).
- Extracted programs are terms in Minlog's internal representation, evaluated via NbE in Scheme.
- Are slow programs the price we have to pay for verified correctness?
- No! I will show you how to reduce Andy's time to 0.340 s, without changing the program.
- The trick is to (automatically) translate the programs into Haskell, which has excellent optimisation support.

Outline

1. Algebras and terms in Minlog
2. Translation into Haskell
3. Back to Andy's SAT solver

Algebras and terms in Minlog

A common extension of Gödel's T and PCF

Simply typed λ-calculus

$+$ Free algebras

$+$ Recursion and corecursion operators

$+$ General recursion with a measure μ

$+$ Program constants

A common extension of Gödel's T and PCF

Simply typed λ-calculus	minimal logic
$+$ Free algebras	(co)inductive predicates
$+$ Recursion and corecursion operators	induction and coinduction
$+$ General recursion with a measure μ	general induction via μ
$+$ Program constants	partial functionals (PCF)

Types

- Simply typed language.
- Base types and function types $\sigma \to \tau$.
- Base types are *free algebras*.
  - Given by (finite) list of constructors (sum-of-products data types).
  - E.g. lists, binary trees:

$$\mathbf{L}_\alpha = \mu_\xi([]^\xi, ::^{\alpha\to\xi\to\xi})$$

$$\mathbf{BinTree}_\alpha = \mu_\xi(\mathsf{Leaf}^{\alpha\to\xi}, \mathsf{Branch}^{\xi\to\xi\to\xi})$$

  - Require at least one constructor without inductive arguments – ensures all algebras are inhabited.
  - Note the type variable α (*polymorphism*).

More on algebras

- Algebras can be simultaneously defined, e.g. finitely branching trees

$$(\mathbf{Ts}, \mathbf{T}) = \mu_{\xi,\zeta}(\mathsf{Empty}^{\xi}, \mathsf{Tcons}^{\zeta\to\xi\to\xi}, \mathsf{Leaf}^{\zeta}, \mathsf{Branch}^{\xi\to\zeta})$$

$$\begin{aligned}
\mathsf{Empty} &: \mathbf{Ts} \\
\mathsf{Tcons} &: \mathbf{T} \to \mathbf{Ts} \to \mathbf{Ts} \\
\mathsf{Leaf} &: \mathbf{T} \\
\mathsf{Branch} &: \mathbf{Ts} \to \mathbf{T}
\end{aligned}$$

More on algebras

- Algebras can be simultaneously defined, e.g. finitely branching trees

$$(\mathbf{Ts}, \mathbf{T}) = \mu_{\xi,\zeta}(\mathsf{Empty}^{\xi}, \mathsf{Tcons}^{\zeta\to\xi\to\xi}, \mathsf{Leaf}^{\zeta}, \mathsf{Branch}^{\xi\to\zeta})$$

$$\begin{aligned} \mathsf{Empty} &: \mathbf{Ts} \\ \mathsf{Tcons} &: \mathbf{T} \to \mathbf{Ts} \to \mathbf{Ts} \\ \mathsf{Leaf} &: \mathbf{T} \\ \mathsf{Branch} &: \mathbf{Ts} \to \mathbf{T} \end{aligned}$$

- Also *nested* definitions are possible:

$$\mathbf{NT} = \mu_{\xi}(\mathsf{Lf}^{\xi}, \mathsf{Br}^{\mathbf{L}_{\xi}\to\xi})$$

More on algebras

- Algebras can be simultaneously defined, e.g. finitely branching trees

$$(\mathbf{Ts}, \mathbf{T}) = \mu_{\xi,\zeta}(\mathsf{Empty}^{\xi}, \mathsf{Tcons}^{\zeta \to \xi \to \xi}, \mathsf{Leaf}^{\zeta}, \mathsf{Branch}^{\xi \to \zeta})$$

$$\begin{aligned} \mathsf{Empty} &: \mathbf{Ts} \\ \mathsf{Tcons} &: \mathbf{T} \to \mathbf{Ts} \to \mathbf{Ts} \\ \mathsf{Leaf} &: \mathbf{T} \\ \mathsf{Branch} &: \mathbf{Ts} \to \mathbf{T} \end{aligned}$$

- Also *nested* definitions are possible:

$$\mathbf{NT} = \mu_{\xi}(\mathsf{Lf}^{\xi}, \mathsf{Br}^{\mathbf{L}_{\xi} \to \xi})$$

More on algebras

- Algebras can be simultaneously defined, e.g. finitely branching trees

$$(\mathbf{Ts}, \mathbf{T}) = \mu_{\xi,\zeta}(\mathrm{Empty}^{\xi}, \mathrm{Tcons}^{\zeta\to\xi\to\xi}, \mathrm{Leaf}^{\zeta}, \mathrm{Branch}^{\xi\to\zeta})$$

$$\begin{aligned} \mathrm{Empty} &: \mathbf{Ts} \\ \mathrm{Tcons} &: \mathbf{T} \to \mathbf{Ts} \to \mathbf{Ts} \\ \mathrm{Leaf} &: \mathbf{T} \\ \mathrm{Branch} &: \mathbf{Ts} \to \mathbf{T} \end{aligned}$$

- Also *nested* definitions are possible:

$$\mathbf{NT} = \mu_{\xi}(\mathrm{Lf}^{\xi}, \mathrm{Br}^{\mathbf{L}_{\xi}\to\xi})$$

- Realizers for simultaneous and nested predicates.

Recursion operators

$$\mathcal{R}^{\tau}_{\mathbf{L}_\alpha} : \mathbf{L}_\alpha \to \tau \to (\alpha \to \mathbf{L}_\alpha \to \tau \to \tau) \to \tau$$

$$\begin{aligned} \mathcal{R}^{\tau}_{\mathbf{L}_\alpha}\ []\ e\ f &= e \\ \mathcal{R}^{\tau}_{\mathbf{L}_\alpha}\ (x{::}xs)\ e\ f &= f\ x\ xs\ (\mathcal{R}^{\tau}_{\mathbf{L}_\alpha}\ xs\ e\ f) \end{aligned}$$

- One for each algebra, parameterised over target type τ.
- Realizer of structural induction.
- Simultaneous algebras use simultaneous recursion operators.
- Nested algebras such as **NT** use *map operators*, e.g.

$$\mathcal{M}^{\sigma\to\rho}_{\lambda_\alpha \mathbf{L}_\alpha} : \mathbf{L}_\sigma \to (\sigma \to \rho) \to \mathbf{L}_\rho$$

Corecursion and destructors

- Dual of recursion and constructors.
- No separate coalgebra – limit-colimit coincidence for domains.
- E.g. destructor for **NT** (here $\mathbf{U} = \mu_\xi(\mathsf{u}^\xi)$ is the unit type):

$$\mathcal{D}_{\mathbf{NT}} : \mathbf{NT} \to \mathbf{U} + \mathbf{L}_{\mathbf{NT}}$$
$$\mathcal{D}_{\mathbf{NT}}\, \mathsf{Lf} \mapsto \mathsf{inl}\ \mathsf{u}, \qquad \mathcal{D}_{\mathbf{NT}}\, (\mathsf{Br}\ as) \mapsto \mathsf{inr}\ as.$$

- Corecursion operator:

$$\begin{aligned} {}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{NT}} &: \tau \to (\tau \to \mathbf{U} + \mathbf{L}_{\mathbf{NT}+\tau}) \to \mathbf{NT} \\ {}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{NT}}\, N\, M &\mapsto \text{case } (M\, N) \text{ of} \\ &\qquad \mathsf{inl}\ \mathsf{u} \to \mathsf{Lf} \\ &\qquad \mathsf{inr}\ qs \to \mathsf{Br}\ (\mathcal{M}^{\mathbf{NT}+\tau\to\mathbf{NT}}_{\lambda_\alpha \mathbf{L}_\alpha}\ qs\ [\mathrm{id}, \lambda_x({}^{\mathrm{co}}\mathcal{R} x M)]) \end{aligned}$$

- Realizer of coinduction.

General recursion with a measure

- General induction with measure $\mu : \tau \to \mathbf{N}$: If $P(x)$ whenever $P(y)$ holds for all y with $\mu(y) < \mu(x)$, then $(\forall x : \tau) P(x)$.

- Realized by general recursion – allowed to make recursive calls on arguments smaller according to μ (ensures termination).

$$^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma} : (\tau \to \mathbf{N}) \to \tau \to (\tau \to (\tau \to \sigma) \to \sigma) \to \sigma$$
$$^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ x\ g = g\ x\ (\lambda_y(\mathrm{if}\ \mu(y) < \mu(x)\ \mathrm{then}\ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ y\ g\ \mathrm{else}\ \mathsf{inhab}_{\sigma}))$$

- Here inhab_{σ} is a canonical inhabitant of type σ – remember all algebras (hence all types) are inhabited.

General recursion with a measure

- General induction with measure $\mu : \tau \to \mathbf{N}$: If $P(x)$ whenever $P(y)$ holds for all y with $\mu(y) < \mu(x)$, then $(\forall x : \tau) P(x)$.
- Realized by general recursion – allowed to make recursive calls on arguments smaller according to μ (ensures termination).

$$ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma} : \overbrace{(\tau \to \mathbf{N})}^{\text{measure}} \to \tau \to (\tau \to (\tau \to \sigma) \to \sigma) \to \sigma $$

$$ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ x\ g = g\ x\ (\lambda_y(\mathrm{if}\ \mu(y) < \mu(x)\ \mathrm{then}\ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ y\ g\ \mathrm{else}\ \mathsf{inhab}_{\sigma})) $$

- Here inhab_{σ} is a canonical inhabitant of type σ – remember all algebras (hence all types) are inhabited.

General recursion with a measure

- General induction with measure $\mu : \tau \to \mathbf{N}$: If $P(x)$ whenever $P(y)$ holds for all y with $\mu(y) < \mu(x)$, then $(\forall x : \tau)P(x)$.

- Realized by general recursion – allowed to make recursive calls on arguments smaller according to μ (ensures termination).

$$
{}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma} : (\tau \to \mathbf{N}) \to \overbrace{\tau}^{\text{input}} \to (\tau \to (\tau \to \sigma) \to \sigma) \to \sigma
$$
$$
{}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ x\ g = g\ x\ (\lambda_y(\text{if } \mu(y) < \mu(x) \text{ then } {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ y\ g \text{ else } \mathsf{inhab}_{\sigma}))
$$

- Here inhab_{σ} is a canonical inhabitant of type σ – remember all algebras (hence all types) are inhabited.

General recursion with a measure

- General induction with measure $\mu : \tau \to \mathbf{N}$: If $P(x)$ whenever $P(y)$ holds for all y with $\mu(y) < \mu(x)$, then $(\forall x : \tau) P(x)$.
- Realized by general recursion – allowed to make recursive calls on arguments smaller according to μ (ensures termination).

$$
{}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma} : (\tau \to \mathbf{N}) \to \tau \to \overbrace{(\tau \to (\tau \to \sigma) \to \sigma)}^{\text{step function}} \to \sigma
$$

$$
{}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ x\ g = g\ x\ (\lambda_y(\mathrm{if}\ \mu(y) < \mu(x)\ \mathrm{then}\ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ y\ g\ \mathrm{else}\ \mathsf{inhab}_{\sigma}))
$$

- Here inhab_{σ} is a canonical inhabitant of type σ – remember all algebras (hence all types) are inhabited.

General recursion with a measure

- General induction with measure $\mu : \tau \to \mathbf{N}$: If $P(x)$ whenever $P(y)$ holds for all y with $\mu(y) < \mu(x)$, then $(\forall x : \tau)P(x)$.

- Realized by general recursion – allowed to make recursive calls on arguments smaller according to μ (ensures termination).

$$ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma} : (\tau \to \mathbf{N}) \to \tau \to (\tau \to \overbrace{(\tau \to \sigma)}^{\text{rec. call}} \to \sigma) \to \sigma $$

$$ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ x\ g = g\ x\ (\lambda_y(\mathrm{if}\ \mu(y) < \mu(x)\ \mathrm{then}\ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ y\ g\ \mathrm{else}\ \mathsf{inhab}_{\sigma})) $$

- Here inhab_{σ} is a canonical inhabitant of type σ – remember all algebras (hence all types) are inhabited.

General recursion with a measure

- General induction with measure $\mu : \tau \to \mathbf{N}$: If $P(x)$ whenever $P(y)$ holds for all y with $\mu(y) < \mu(x)$, then $(\forall x : \tau) P(x)$.

- Realized by general recursion – allowed to make recursive calls on arguments smaller according to μ (ensures termination).

$$
\begin{aligned}
&{}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma} : (\tau \to \mathbf{N}) \to \tau \to (\tau \to (\tau \to \sigma) \to \sigma) \to \sigma \\
&{}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ x\ g = g\ x\ (\lambda_y(\mathrm{if}\ \mu(y) < \mu(x)\ \mathrm{then}\ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ y\ g\ \mathrm{else}\ \mathsf{inhab}_{\sigma}))
\end{aligned}
$$

- Here inhab_σ is a canonical inhabitant of type σ – remember all algebras (hence all types) are inhabited.

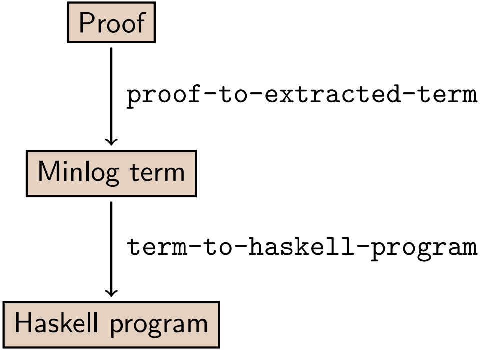
Program constants

- The user can add their own constants – this is the PCF part.
- Defined by pattern-matching – no requirement of exhaustive patterns or recursive calls only on smaller arguments.
- User is asked to prove totality, but this can be skipped.
- Semantics using domains (in the form of Scott's *information systems*).
- E.g. parity : $\mathbf{N} \to \mathbf{B}$

$$\begin{array}{lccl} \text{parity} & 0 & = & \text{F} \\ \text{parity} & (\text{Succ } 0) & = & \text{T} \\ \text{parity} & (\text{Succ } (\text{Succ } n)) & = & \text{parity } n \end{array}$$


Translating into Haskell

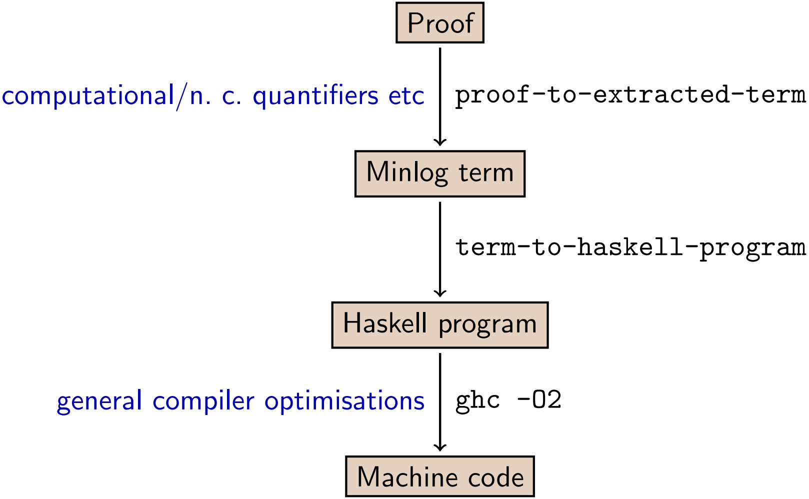
From proof to program



From proof to program

$$\begin{tikzcd}
\boxed{\text{Proof}} \arrow[d, "\texttt{proof-to-extracted-term}"] \\
\boxed{\text{Minlog term}} \arrow[d, "\texttt{term-to-haskell-program}"] \\
\boxed{\text{Haskell program}} \arrow[d, "\texttt{ghc -O2}"] \\
\boxed{\text{Machine code}}
\end{tikzcd}$$

From proof to program



Minlog types to Haskell types

- Translation Minlog types $\to$ Haskell types straightforward.
- Algebras mapped to data types. For “builtin” types:

Minlog type	Haskell type
$\mathbf{N}$, $\mathbf{Z}$, $\mathbf{P}$	`Integer`
$\mathbf{Q}$	`Rational`
$\mathbf{B}$	`Bool`
$\mathbf{L}_\alpha$	`[`α`]`
$\mathbf{U}$	`()`
$A + B$	`Either A B`
$\mathbf{U} + A$	`Maybe A`
$A \times B$	`(A, B)`
$A \to B$	`A -> B`

- Notable exception: **R** treated like any other algebra – no direct Haskell equivalent (certainly not `Float`).

Other algebras

- Other algebras translated – straightforward since given by constructors in both Minlog and Haskell.
- Haskell supports both mutual and nested data types.
- Add `deriving (Eq, Show, Read, Ord)` for finitary algebras.
- Need to make sure that data type and constructor names start with a capital letter.

Generating a Haskell program

Given a list of terms $\vec{t}$:

1. Recursively find all program constants, operators and their types occurring in $\vec{t}$.
2. Generate data type declarations, and functions for operators and program constants.
3. Translate the terms in $\vec{t}$ themselves.

Translating terms

- Mostly straightforward.
- Translate variables to variables, lambda terms to lambda terms etc.
- Minlog has already taken care of making variables non-clashing (via α-conversion).
- However, Minlog is fond of variable names such as

  ```
  (integer=>(integer@boole)=>nat)_0
  ```

  which are not valid Haskell names (and long!).
- We make sure all illegal characters are removed.
- Replace with shorter names, unless the name was chosen by the user.

Recursion operators

- For recursion operators, we construct Minlog terms

$$r_i := \mathcal{R}_\sigma^\tau\ (c_i\ \vec{t})\ \vec{e}$$

  with fresh variables $\vec{t}$ and $\vec{e}$ for each constructor c_i of σ.

- We then normalize the Minlog terms in Minlog $\rightsquigarrow \mathtt{nt}(r_i)$.

- Generate a Haskell function defined by

$$r_0 = \mathtt{nt}(r_0)$$
$$\dots$$
$$r_k = \mathtt{nt}(r_k)$$

- Ensures that Haskell semantics coincide with Minlog semantics.

```
listRec : [a] -> b -> (a -> [a] -> b -> b) -> b
listRec [] e f = e
listRec (x : xs) e f = f x xs (listRec xs e f)
```

Corecursion operators

- For corecursion, no distinction is made between different constructors.

- Minlog has a function to expand a corecursion constant once (Scheme and Minlog are strict languages).

```
nTCoRec : b -> Maybe [Either NT b] -> NT
ntCoRec n m =
 case (m n) of
   Nothing -> Lf
   (Just w) -> Br (fmap (\ y -> (case y of
                                   Left h -> h
                                   Right e -> nTCoRec e m) w))
```

- Map operators translated to `fmap` from `Functor` type class – can be derived automatically by GHC using the `DeriveFunctor` flag.

Program constants

- Program constants are basically Haskell pattern matching functions.
- Complication: we translate natural numbers to integers, but cannot pattern match on integers as natural numbers.
- Solution: use Haskell's guard conditions.

```
parity :: Integer {-Nat-} -> Bool
parity 0 = False
parity 1 = True
parity n | n > 1 = parity (n - 2)
```

- Similar considerations for **P** and rational numbers.

General recursion with a measure

$$ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma} : (\tau \to \mathbf{N}) \to \tau \to (\tau \to (\tau \to \sigma) \to \sigma) \to \sigma $$
$$ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ x\ g = g\ x\ (\lambda_y(\mathrm{if}\ \mu(y) < \mu(x)\ \mathrm{then}\ {}^{\mathrm{g}}\mathcal{R}^{\tau}_{\sigma}\ \mu\ y\ g\ \mathrm{else}\ \mathsf{inhab}_{\sigma})) $$

- Two options: same behaviour as Minlog or taking advantage of laziness.
- Minlog evaluates the measure at each recursive call – expensive.
- Stops non-terminating evaluation where the body is infinitely expanded (Minlog and Scheme strict languages).

General recursion with a measure (cont.)

- Translation offers to skip the check – gives another realizer that is still sound. (Controlled by `HASKELL-GREC-MEASURE-FLAG`.)

```
gRec :: a -> (a -> (a -> b) -> b) -> b
gRec x g = g x ( y -> gRec y g)
```

- However, now the link to Minlog semantics is lost: ${}^{\mathrm{g}}\mathcal{R}_{\sigma}^{\tau}$ is always total in Minlog, modified version not necessarily so in Haskell.

```
  gRec 0 (\ y h -> h y)
= (\ y h -> h y) 0 (\ z -> gRec z (\ y h -> h y))
= (\ z -> gRec z (\ y h -> h y)) 0
= gRec 0 (\ y h -> h y)
= ...
```

(e.g. with identity measure $\mu : \mathbf{N} \to \mathbf{N}$)

Canonical inhabitants

- Previous slide used the canonical inhabitant inhab_σ.
- Also used to realize *ex-falso-quodlibet* $\bot \to A$.
- Was okay since all Minlog types are inhabited by total elements – not true for Haskell!
- Solution: introduce a type class

```
class Inhabited a where
      inhab :: a
```

Canonical inhabitants (cont.)

```
class Inhabited a where
      inhab :: a
```

- Now we need to track inhabitedness constraints and add them to type signatures.
- Can be complicated with mutually recursive calls etc – fixed point algorithm.
- Also need to generate instances for concrete types τ that use inhab_{τ}.

Back to Andy’s SAT solver

Extracting a DPLL solver

- Andy gave me his Minlog development for the DPLL solver.
- I extracted his program and wrote 30 lines of Haskell.
  - Get file name from command line, use a library to parse input in the DIMACS format (15 lines).
  - `Show` instances for non-finitary data types (15 lines).
- Using Haskell's laziness, we can write a `Show` instance so that we only calculate YES/NO (satisfiable), without a witness.

Benchmark

Formula	Minlog	Interpreted (`ghci`)		Compiled (`ghc -O2`)	
	Witness	Witness	Yes/No	Witness	Yes/No
PHP(4,3)	15.32s	0.17s	0.12s	0.008s	0.004s
PHP(4,4)	6.87s	0.08s	0.07s	0.000s	0.000s
PHP(5,4)	219.78s	1.52s	1.08s	0.032s	0.020s
PHP(5,5)	33.15s	0.18s	0.19s	0.004s	0.004s
PHP(6,5)	2245.27s	16.68s	11.71s	0.340s	0.124s
PHP(6,6)	84.88s	0.54s	0.53s	0.012s	0.012s

Thanks!

`term-to-haskell-program` is available in the SVN ("latest") version of Minlog.