Quick and Reusable Code Generation for Idris
Integrating dependent types into the industrial services

Taine Zhao & Yukiyoshi Kameyama

Computer Science, University of Tsukuba

`thaut@logic.cs.tsukuba.ac.jp; kam@cs.tsukuba.ac.jp`

Abstract

As a dependently-typed functional programming language, Idris shows quite a high expressiveness with its type system under a considerably strong static guarantee, which is capable of ending many difficult and frequently occurring bugs in the industrial world.

To leverage its powerful static programming language features and make existing industrial applications safer and more expressive, a valid approach is to implement code generation back ends for Idris.

Whereas Idris has already provided convenient interfaces to support agnostic back ends, it is still cumbersome to import Idris programs into an existing programming language. As the existing programming languages are already in such a quantity and continuously proliferating, and many programming languages are quite similar, the implementations of different Idris back ends can have quite a lot of overlaps and redundancies.

To address them, we introduce a "common" intermediate representation, Weakest Declaration, which is a much simpler IR and contributes to reusability of analyses and transformations required by different Idris back ends. As a result, we allow making an Idris back end in a most simplified routine, which can usually be accomplished within very few code in few hours or, even minutes.

Background

Idris is a programming language with dependent types, hence it is capable of ending many difficult and frequently occurring bugs in the industrial world: For instance, dependently typed programs can statically verify the validness of array indexing [9], or statically verify the validness of tensor operations [7] [6], especially statically checking/inferring tensor dimensions.

```
app : Vect n a → Vect m a → Vect (n + m) a
app Nil        ys = ys
app (x :: xs) ys = x :: app xs ys

test : Integer
test = 2 `Vect.index` ([1, 2] `app` [4])
✓

test : Integer
test = 3 `Vect.index` ([1, 2] `app` [4])
✗ 3 is not strictly less than 2 + 1!
```

Figure 1: Canonical example, a dependent vector

To leverage its advanced type system in practice, a valid approach is implementing code generation back ends for Idris, and then integrating Idris into the existing industrial applications.

To achieve this, Idris has already paid lots of efforts to their code generation facilities, and finally provided a convenient interface for back end plugins [4].

We hereby provide the definition of Idris defunctionalised IR with some details omitted and simplifications, hereafter as $DDecl$.

⟨expr⟩ ::= *Var name*
| *App bool name [expr]*
| *Let name expr expr*
| *Update name expr*
| *Proj expr int*
| *Cons name [expr]*
| *Case expr [alt]*
| *Const constant*
| *Foreign ...*
| *Op primitive-op [expr]*
| *DoNothing*
| *Error string*

⟨decl⟩ ::= *DefFun name [name] expr*
| *DefCons name int*

⟨alt⟩ ::= *ConCase name [name] expr*
| *ConstCase constant expr*
| *DefaultCase expr*

⟨arith-type⟩ ::= *float* | *int*
⟨primitive-op⟩ ::= *+ arith-type*
| *- arith-type*
| ** arith-type*
| *sdiv arith-type*
| *udiv arith-type*
| *...*

Language 1: Defunctionalised Declarations

- $Cons$, $DefCons$: constructing tagged unions, and constructor definitions
- $Case$: pattern matching, or deconstructions
- $Proj$: projections, on tuples and tagged unions
- $primitive\text{-}op$: $+, -, *, /$, and other primitive operators defined and used by Idris compiler

Investigation

$DDecl$ is already convenient for code generation, however still **overly high level**, and could produce **redundant repetitions** when comparing the implementations of many different back ends.

$DDecl$ has Let expressions which is responsible for variable introductions and **name shadowing**, however missing in most older programming languages like C/C++, Java, Ruby, Python, etc.

There's two ways of eliminating Let expressions(see **figure 2,3,4**). Using IIFE is straightforward, however will extremely **SLOW** down the back ends of dynamic programming languages. Another approach requires analyzing the scope of your programs and, performing register allocation optimizations to avoid using too many local variables.

Besides, $DDecl$ has algebraic data types(abbr. ADT), and the underlying of ADTs is the representation of sum types and product types(see **figure 4, 5**).

The product type merely requires the back end to implement tuples, which is usually straightforward. However, the sum type, or tagged union, requires the back end to implement tags. A tag shall be $O(1)$ comparable, and identical to the qualified name of the data constructor, which is supported by programming languages with $Symbol$ types. When the back end has no native $Symbol$ type, our compiler should be responsible for emulating symbols.

There's other language constructs too high level, or difficult to achieved by many existing languages, in the case, our compiler shall desugar or lower them:
- pattern matching
- block expression [1] [2]
- arbitrary identifier
- tail call
- mutation

```
let x = val1 in
let x = val2 in
func(x)
```

Figure 2: *Let* in *DDecl*

```
x0 = val1
x1 = val2
func(x1)
```

Figure 3: Eliminating *Let* by name mangling

```
func _(x){
    return func _(x){
        func(x)
    }(val2)
}(val1)
```

Figure 4: Eliminating *Let* by immediately invoked functions(abbr. IIFE)

```
Nil = 'Nil
func Cons(head, tail){
    return ('Cons, head, tail)
}
```

Figure 5: ADT internals in back ends

```
data List a = Nil | Cons a (List a)
```

Figure 6: Algebraic Data Types(ADTs)

Other than above desugaring and lowering, $DDecl$ is still tough to tackle as back end implementer should aware various primitive operations and language constructs, such as projections $Proj$, FFIs $Foreign$, throwing exceptions $Error$, and primitive operators Op.

It could be beneficial to transfer the complexity to the upstream, i.e., a runtime system of the back end [3], see columns $DDecl$ and *JavaScript-like pseudo code* in **Table 1**.

Cases	$DDecl$	JavaScript-like pseudo code	$WDecl$
Integer Additions	Op (+) [a, b]	RTS.op_plus(a, b)	Ext (ExtApp "op_plus" [a, b])
Projections	Proj a 1	RTS.proj(a, 1)	Ext (ExtApp "proj" [a, 1])
Pattern Matching 1	Case a [ConstCase 1 233]	switch (a){ case 1: 233 }	...
Pattern Matching 2	Case a [ConCase "Cons" [_, _] 321]	switch (RTS.proj(a, 0)) { case 'Cons: 321}	...
Construction 1	Cons "Nil" []	'Nil	Const (Symbol "Nil")
Construction 2	Cons "Cons" [1, a]	RTS.make_tuple('Cons, 1, a)	...

Table 1: Correspondences between different program representations

Proposal

To address problems mentioned in section $Investigation$, We hereby propose a new IR, which is lightweight, neat and compact, and finally capable of getting used to implement a back end reasonably fast.

⟨block-stmt⟩ ::= *[stmt]*
⟨stmt⟩ ::= *Intro name*
| *Up name expr*
| *If expr block-stmt block-stmt*
| *Eff expr*
| *Ret expr*
| *Switch expr [(constant , block-stmt)] block-stmt*

⟨ext⟩ ::= *ExtApp name [expr]*
| *ExtVar name*
⟨expr⟩ ::= *Var name*
| *App name [expr]*
| *Const constant-literal*
| *Ext ext*

Language 2: Weakest Declarations

Instead of leaving a blank for the internal implementations of product types, FFIs, constructors of algebraic data, we can assume a generally applicable implementation for each of them, and finally, transform $DDecl$ to the proposed IR, with desugaring Let expressions and block expressions, and permitting optional transformations which might be required by some back ends.

We'd call it a **Weakest Declaration**(abbr. $WDecl$), as it is weak enough to be expressed by all of the target programming languages investigated in this project.

See **Table 1** for the correspondences between $DDecl$, JavaScript-like pseudo code, and $WDecl$.

Results code at https://bitbucket.org/thautwarm/ppl2020-idris-codegen-backend/src/master

We implemented the transformation from $DDecl$ to $WDecl$ in the Haskell side, then produced a binary executable which compiles Idris source files to a standalone **.qb** file.

A target language will later read $WDecl$ IR from the **.qb** file, and then do back end specific code generation, i.e., Haskell side is not responsible for generating executable code from Idris source files.

This strategy is advantageous, as most long-living languages have their own libraries or features to idiomatically manipulate their own programs as data.

Finally, we provided back end implementations for Python, Julia and Ruby, with a surprisingly concise code, each of which consists of only trivial procedures within a sparse 200 lines, and tested by the examples provided by the official website of Idris tutorials. We call our design and implementation **Quick Backend**.

Target Language	Existing Implementation, Line Count	Line Count by QB(abbr. Quick Backend)	Notes
Python2	ziman/idris-py, > 900	177	QB: Python 2+3
Python3	thautwarm/idris-python, > 1000	177	ditto
Ruby	mrb/idris-ruby, > 220	209	QB: full-featured + optimized
Julia	none so far	171	very fast

Table 2: Line count comparisons of results

Note that, by using the Quick Backend approach, a lot of stuffs, such as FFI, primitive operations get decoupled from the back end implementation itself, as a result, we gain extensibility, and further optimizations can be easily introduced as plugins.

Conclusions

We achieved our goal of implementing Idris back ends quickly, with the capability of **reusing** many transformations for multiple back ends.

Some available transformations:
- LISP-symbol emulations
- arbitrary identifier to the classic Java identifier
- block expression desugaring
- let expression desugaring
- pattern matching compilation

Due to the limitations of our bandwidth, some more difficult transformations are missing. However, it shall not be a problem, as what matters here is how to reuse of transformations for multiple back ends.
Some WIP transformations:
- tail call scheduling
- register allocation optimizations
- mutations to state monads
- garbage collections

Forthcoming Research

Although Idris is powerful enough to express very complex static properties, still are there cases runtime checking is in need.
Some reasons here could be

- Constructing proofs for complex properties is pretty niche, requiring specific knowledge about dependent types and theorem proving.
- Programmers incapable of proving the correctness with their code, might be able to prove it in other means.
- Idris itself is not perfect enough to prove things as it is, i.e., hand-written mathematical proofs can sometimes be easier.

To support runtime checking, the reasonable error reports and debugging facilities shall be provided, which both require some metadata from the original source code, e.g.,

- filename
- source line number
- source column number

All these metadata are missing in $DDecl$ or other convenient IRs provided by Idris, as a consequence, it's impossible to get a practical runtime checking. Hence, how to access metadata of Idris programs could be a future research.

Acknowledgements

References

[1] GCC, the gnu compiler collections, statement expressions. http://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html. Accessed: 2020-02-25.

[2] PEP572: Assignment expressions. https://www.python.org/dev/peps/pep-0572. Accessed: 2020-02-25.

[3] Andrew W Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–380, 1990.

[4] Edwin Brady. Cross-platform compilers for functional languages.

[5] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552–593, 2013.

[6] Tongfei Chen. Typesafe abstractions for tensor operations (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, pages 45–50, 2017.

[7] Frederik Eaton. Statically typed linear algebra in haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 120–121, 2006.

[8] Archibald Samuel Elliott. *A concurrency system for IDRIS & ERLANG*. PhD thesis, Bachelors Dissertation, University of St Andrews, 2015. URL https://lenary.co.uk/publications/Elliott_BSc_Dissertation.pdf, 2015.

[9] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, 1998.