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 a large amount and continuously proliferating, and the implementations of different back ends certainly have quite a few overlaps.

To address this, 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 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 [8], or statically verify the validness of tensor operations [6] [5], especially statically checking/inferring 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 integrating Idris into the existing 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 [3].

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*
| ...

- $Cons$, $DefCons$: 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

Introduction

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

Firstly we check Let expressions. They are responsible for variable introductions, and also capable of **shadowing variables**, but unfortunately missing in most old programming languages such as C/C++, Java, Ruby, Python, etc.

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

Figure 2: let in $DDecl$

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

Figure 3: Eliminating let by name mangling

```
fun tmp(x) {
    fun tmp(x) {
       func(x)
    }(val2)
}(val1)
```

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

Eliminating let by IIFE is very handy without requiring much code, however extremely **SLOW** down the back ends of dynamic programming languages.

As for name mangling,

- renaming variables itself needs a simple pass to analyse the scope of your program!
- considering register allocation problems?

```
let x = val1
let x = func1(x)
in   func2(let x = val2 in func3(x))
   ; func4(x)
```

Figure 5: Occurrences of distinct x

```
set x0 = val1
set x0 = func1(x0)
in   func2(set x1 = val2 in func3(x1))
   ; func4(x0)
```

Figure 6: Register allocation optimisation for x

Besides, the underlying of ADTs is the representation of the tagged unions.

A valid approach is, firstly emulate LISP symbols to achieve $O(1)$ comparable **tags**, and then use tuples whose 1st element is a LISP symbol, to represent ADTs.

```
data [a] = Nil | Cons a [a]
```

Figure 7: Algebraic Data Types(ADTs)

```
lst1 = 'Nil // or ('Nil, ) ?
lst2 = ('Cons, head, tail)
```

Figure 8: ADT internals in back ends

We also have to

- translate pattern matching things to non-pattern match languages
- translate block expressions [1] [2] into languages whose expressions cannot accommodate statements
- support primitive operations
- support FFI

Options shall be provided here to control the properties of the IR, to achieve the reuse of

- desugaring block expressions, which may requires register allocation optimisations.
- desugaring pattern matching to switch statements and if statements
- desugaring data constructors to normal functions
- using LISP-style $Symbol$ as the tag of tagged union, for a language with $Symbol$ data.
- emulation of $Symbol$ in the language without $Symbol$ data for tags of tagged unions

Those stuffs are not difficult, but quite annoying when implementing them again and again, for each back end.
What is the possibly simplest IR for code generation?

Proposal

To address problems mentioned above, We hereby propose an IR, which is lightweight, neat and compact, and finally capable of getting used to support a back end quickly.

⟨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*

Figure 9: Weakest Declarations

Instead of leaving a blank for the internal implementations of tuples, FFIs, constructors of algebraic data, we can assume a generally applicable implementation, and transform $DDecl$ to the proposed IR, which we'd call it a **Weakest Declaration**(abbr. $WDecl$).
$WDecl$ desugars Let expressions to $Update$ statements, simplifies function arguments to accept variables or constants only, and avoids requirement of expressing block expressions in the target language.

Further, we point out some correspondences between the original $DDecl$ and our $WDecl$.

Cases	*DDecl*	*WDecl*
Addition Integers	Op (+ int) a b	App "+"[int , a, b])
Projections	Proj a 1	App "proj"[a, 1]
Pattern Matching 1	Case a [ConstCase 1 233]	Switch a [(1, 233)]
Pattern Matching 2	Case a [ConCase "Cons"[_, _] 321]	Switch App "proj"[a, 0] [("Cons", 321)]
Construction 1	Cons "Nil"[]	App "make_symbol"["Nil"]
Construction 2	Cons "Cons"[1, a]	App "make_tuple"[App "make_symbol"["Cons"], 1, a]

In the above table, for being concise, we omitted the constructions of Const, but simply use literals like 1, 321, "+" instead.

Results

We implemented the transformation from $DDecl$ to $WDecl$ in the Haskell side, produce it a binary executable, which implements the Idris back end interfaces.
Hence, we become capable of compiling an Idris project, fetching its $WDecl$ and dumping the IR to the disk.
As a target language, it'll read the standalone file of $WDecl$ IR, parse it, and then do back end specific code generation, which means that the Haskell side is not responsible for generating executable code.
This is advantageous as each long-living language has its own libraries or features to manipulate their own program as data.
TODO
TODO
TODO
TODO
TODO
TODO
TODO: show various back ends by $WDecl$, report the size of codebase for each implementation.

Conclusions

TODO

Forthcoming Research

Although Idris is powerful enough to express very complex static properties, there're still cases where runtime checking will be needed.
Some reasons here could be

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

To support runtime checking, the reasonable error reports and debugging shall be supported, 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.

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

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

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

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

[7] 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.

[8] 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.

Acknowledgements

Archibald Samuel Elliott for his elaborated notes about Idris $DDecl$ [7], and so far all those IRs are highly-undocumented in Idris official site.

Contact Details

Taine Zhao - thaut@logic.cs.tsukuba.ac.jp
Yukiyoshi Kameyama - kam@cs.tsukuba.ac.jp

Supplementary

$WDecl$ is actually simpler than the so-called $SDecl$ provided by the Idris compiler [4] as their simplest IR.
The reason why we take $WDecl$ as the weakest is, incidentally, it's like a simplified form of ASTs of the Python Programming Language. Python is the weakest programming language among all well-known dynamic programming languages, which is to say, despite of the details of object models/systems, Python can be trivially expressed/translated to Ruby, Lua, JavaScript, Erlang, etc., whereas transforming from the latter ones to Python is thorny, due to the lack of block expressions and assignment expressions [2]. Although the weakness of Python is considered harmful in regular programming tasks, it unveils what a generally transformable upstream IR shall look like.