Simon Dobson

Systems Research Group
School of Computer Science and Informatics
UCD Dublin Belfield, Dublin 4, Ireland
http://www.ucd.ie/csi
simon.dobson@ucd.ie

The De Bruijn principle and the compositional design of programming languages

Overview

While it's recognised that programming languages exert a huge influence on the way we attack problems, they remain difficult to build

- Large investment of time and effort, large learning curve

We can apply the principles of component-based development to languages themselves

- Re-use, differentiate, low-overhead experimentation
- Simplify dissemination
- Integrate other artifacts such as proofs or theorems

My goal: describe some early work on modular, compositional languages and their manipulation

Our specific domain in SRG

SRG does work on software technology, but with one specific target of pervasive systems

- Sensor-intensive systems
- Programming with uncertainty

Both of these areas could beneficially use new programming approaches

- Integrate uncertain reasoning into the control structures
- Want to experiment with new constructs, but without sacrificing links to other infrastructure (typically in Java), without excessive overhead, and maintaining lightweight structure
- More generally, a useful way to experiment with software technology

Previous work

Classical language development

- Dragon book, GNU tool-chain, …
- DIY interpreters in functional languages, …
- Rewrite systems, proof systems, …
- Typically define languages in a chunk, not always extensible

Compositional language development

- Vanilla – compose components written in Java
- Components coded, not declared
- Intensional programming and .NET

Dobson *et al.* Vanilla: an open language framework. Proc. GCSE'99.

Lack of declarative definition of features weakens our ability to analyse or link t more advanced analysis tools

De who?

The De Bruijn principle (or criterion) is a fairly common principle of proof systems

> The correctness of a system as a whole depends upon the correctness of a small core, that is then used to construct the rest of the system

The same argument sometimes applies to ordinary software or languages

- Get a core correct, use that to define other elements correctly
- Can lead to an uncomfortable awkwardness in (at least) syntax
- Selection of a core can be a bit arbitrary and/or unsuitable

What goes into a language?

When defining languages we typically view them as a collection of relations and other elements

- Abstract syntax
- Concrete syntax
- `type-of` and `subtype-of` relations – typing, static semantics
- `reduces-to` relation – dynamic semantics
- Other relations (information flow, side effects, …)

Language research typically takes these elements as a base and then tweaks them by adding new abstract syntax, parallel definitions of properties etc

- Structural induction generally used throughout – remarkably few properties don't follow the abstract syntax

Insights – feature independence

Many features can be defined largely independently

- Minimal constraints on sub-terms

For example, an `if` expression

- Abstract syntax `(if c t f)`
- `c` must type as a boolean expresion
- `t` and `f` must have a common supertype, which is the type of the `if` expression itself

Concrete syntax is typically much less reusable, but is largely ephemeral anyway

It suffices that expressions *can* be typed/evaluated: we don't need to constrain the details

- Re-use the definition of `if` in any language that wants it
- …or remove it from languages in which it's inappropriate

Fixpoints

To define `if` we need to "import" *bool*

Overall `type-of` is the fixpoint of the imports of all the rules in the language

EXP IF

$$\frac{\Gamma \vdash c : bool \qquad \Gamma \vdash t : X \qquad \Gamma \vdash f : X}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ f : X}$$

We can define each piece of the `type-of` relation independently in terms of the types of other terms

Need care in dealing with order of matching, double-evaluation etc

VAL IF TRUE

$$\frac{\Gamma \vdash c \rightsquigarrow true \qquad \Gamma \vdash t \rightsquigarrow v}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ f \rightsquigarrow v}$$

VAL IF FALSE

$$\frac{\Gamma \vdash c \rightsquigarrow false \qquad \Gamma \vdash f \rightsquigarrow v}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ f \rightsquigarrow v}$$

Same applies to `reduces-to`

Insights – commonality

There is a huge similarity between even superficially very different languages

- E.g. lazy functional languages add thunks and change the reduction order on functions
- …and otherwise behave remarkably similarly to eager languages
- …and share many features with object-oriented languages
- …and lazy lists are interesting *per se*, not just in conjunction with normal-order reduction

This suggests that *feature sets* rather than languages are the real loci of innovation

- Does our focus on language innovation comes from the need to build interpreters and compilers *ex nihilo*, rather than from any fundamental coupling of the elements?

Insights – blurred boundaries

The boundary between “compile-time” and “run-time” comes down to the use of static relations (`type-of`) in dynamic relations (`reduces-to`)

- Dependent types, run-time code inclusion, …
- If any rule does this, the language needs (at least some) dynamic type information

Reify any metalanguage feature into the language

Properties such as static checking, maximum types and strong normalisation depend critically on feature sets chosen

We'll come back to this later

Nirvana – goals

Nirvana is an experimental toolkit and workbench for compositional language design

Goals

- Compositional definition – separate issues and compose
- Close links to other artifacts – especially proofs
- Libraries of feature sets – reuse base definitions, focus on what's different in the language under development
  - *While remaining within the standard framework of language design as far as possible*
- Efficient and lightweight implementation – never be as good as a dedicated compiler, but aim for enough for real evaluation

Written in Scheme, portable and lightweight

- Lightweight = results can be targeted at a microcontroller

Feature sets

Nirvana divides the language design space into feature sets

- Value domains, domain values, relations, abstract syntax, type rules, reduction rules, primitives

A given feature set defines a set of features, generally composing other features

- Each feature set should focus on a single set of definitions that are "atomic" in the sense of not being meaningfully decomposable into smaller feature sets
- Typically this means all the rules etc relating to a given fragment of abstract syntax

There are exceptions, for example separating the syntax and typing of functions from their reduction rules to support eager and lazy reduction

Feature sets – example

```
(define-feature-set if "http://www.programming-nirvana.org/features/simple-if"
 (uses (feature-set "http://www.programming-nirvana.org/features/standard")
       (feature-set "http://www.programming-nirvana.org/features/boolean-arithmetic"
                    boolean))

 (ast (if c t f))

 (rule exp-if "Conditional expression"
       (type-of env (if c t f) e)
       (type-of env c boolean:boolean-type)
       (type-of env t e)
       (type-of env f e))

 (rule val-if-true "Conditional on true"
       (reduces-to env (if c t f) tv)
       (reduces-to env c boolean:true-literal)
       (reduces-to env t tv))
 (rule val-if-false "Conditional on false"
       (reduces-to env (if c t f) fv)
       (reduces-to env c boolean:false-literal)
       (reduces-to env f fv)))
```


Feature sets – global namespace

```
(define-feature-set if "http://www.programming-nirvana.org/features/simple-if"
 (uses (feature-set "http://www.programming-nirvana.org/features/standard")
       (feature-set "http://www.programming-nirvana.org/features/boolean-arithmetic"
                    boolean))
…)
```

Each feature set is given a globally-unique URI

- Same approach as for RDF ontologies

Import feature sets required

- Prefix tags to ensure uniqueness
- Tag `boolean:true` expands to feature identifier URI http://www.programming-nirvana.org/features/boolean-arithmetic#true, which is a globally unique identifier for that value
- …and similarly for abstract syntax, relations etc
- `standard` feature set defines `type-of` etc

Feature sets – rules

```
…
(ast (if c t f))

 (rule exp-if "Conditional expression"
       (type-of env (if c t f) e)
       (type-of env c boolean:boolean-type)
       (type-of env t e)
       (type-of env f e))
…
```

Introduce a AST node with given arity

- Each AST node is globally unique

Define individual rule (conclusion followed by hypotheses) to be added to global relation definition

- Recursive calls to relation
- Single-assignment and unification
- Code generator builds Scheme functions for each relation

Optimised for the relative lack of backtracking in most cases

Languages

```
(define-language simple
  "http://www.programming-nirvana.org/languages/simple"
  "A simple language, used for demonstration"
  "http://www.programming-nirvana.org/features/standard"
  "http://www.programming-nirvana.org/features/binding"
  "http://www.programming-nirvana.org/features/integer-arithmetic"
  "http://www.programming-nirvana.org/features/boolean-arithmetic"
  "http://www.programming-nirvana.org/features/if"
  "http://www.programming-nirvana.org/features/simple-function"
  "http://www.programming-nirvana.org/features/simple-function-normal-order")
```

Load all the feature sets from the internet to construct an interpreter

- This is a canonical definition of a particular language, in the sense that it is absolutely unambiguously different to any other language that might use similar syntax

Nirvana has support for off-line operation, re-direction of URIs etc - the feature set identifiers can be re-mapped to other sites without destroying the uniqueness properties

Abstract syntax *via* XML

Although we've used Scheme syntax, it's obviously trivial to use XML instead

- Use feature set identifiers as namespaces
- An XML processor will expand the tags into full feature URIs

A language is then described using a collection of XML documents defining its individual feature sets

- Nirvana will walk the hyperstructure to construct a language tool

Don't get much gain from the use of XML over Scheme, other than the obligatory buzzword-compliance…

Back to De Bruijn

The properties we want to prove about languages typically follow the inductive structure of the rules

…at least for many properties, at least when sensibly defined, …

- Each feature set can provide its own contribution to the overall correctness
- Many feature sets will have no impact on many properties, for example `if` doesn't affect strong normalisation

Correctness of a language depends on a small(er) set of feature sets and their properties

- Get correctness and proof by composition as far as possible
- Generally whole-language, but some additions may be known to be property-preserving

Compositional correctness

We conjecture – and it's no more than that yet – that we can inject properties from feature sets into a proof system

- Correctness of the language
- Correctness of applications built with languages – pass proof artifacts out into applications to aid proofs
- Are some languages "intrinsically provable"?

A parallel relation carrying proofs

- The constructive type theory view alongside the more implementational view
- Kick out proof obligations for the designer

Current state and future work

Nirvana is currently on internal alpha-test

- Basic infrastructure and workbench (as Scheme macros)
- Small set of feature sets defined
- Some optimisation (De Bruijn indices – yep, him again…)

Target is to get a public beta before Christmas

- A more complete collection of feature sets (functional languages, both Java-style and Abadi/Cardelli-style objects, …)
- Sub-structural features for linear types etc
- Generalised aspect weaving (Walker's aspect calculus)
- Simple link to PVS for proofs

What follows from the language structure?

Conclusion

Compositional language design

- Build components and then compose
- Re-use feature sets where possible, focus in what's novel
- Global namespace, web-accessible, construct tools on the fly
- Link to proofs of properties/applications

An early prototype demonstrates utility of the approach, but still needs significant work

Applications in pervasive computing and mobile systems

- Novel language structures
- Trust maintenance, proof-carrying code
- Insight into what's core in language design and use