Programming Language Technology for Niche Platforms

Tero Hasu

BLDL and University of Bergen

Bergen, 3 March 2017

more markets, more opportunities

BlackBerry App World Generates Highest Revenue Per App

Posted on February 28, 2011 by Jeff Bacon

	2010 Revenue (Millions of U.S. Dollars)		~Avg. # Apps	$/App	vs. Apple
Apple	iTunes App Store	$ 1,782	275000	$ 6,480.00	-
RIM	BlackBerry App World	$ 165	18000	$ 9,166.67	41%
Nokia	Ovi Store	$ 105	16000	$ 6,562.50	1%
Google	Android Market	$ 102	85000	$ 1,200.00	-81%

Source: Bacon on the Go (http://casualbits.wordpress.com) by Jeff Bacon, Revenue via IHS Screen Digest February 2011

motivation

- readiness to pursue niche platform opportunities

chosen strategy

- have multi-platform software production tooling built around an adaptable programming language capable of existing in variations of itself

thesis' contributions

- technologies
  - and suggestions for applying them

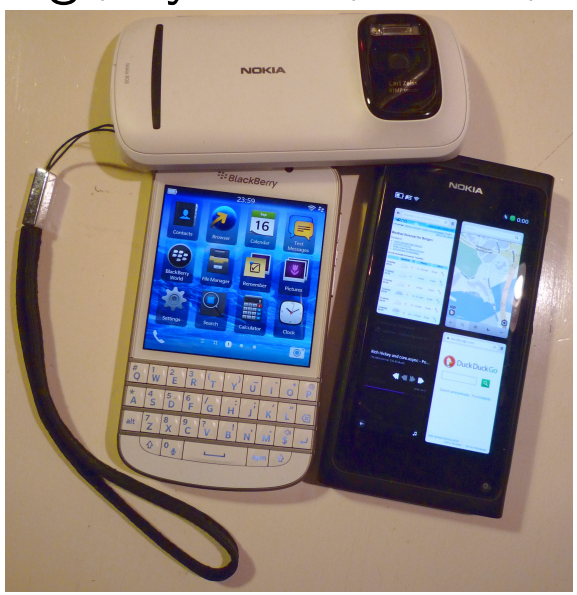
overview

1. niche platforms
2. strategy
3. technologies / papers (chapters 2–5)
   - Source-to-Source Compilation via Submodules
   - Illusionary Abstract Syntax
   - Inferring Required Permissions for Statically Composed Programs
   - Declarative Propagation of Errors as Data Values

part 1: niche platforms

niche platform, defined

- a *platform*:
  - software can be written for it, and run on it
- a *niche platform*: one without a large developer ecosystem
  - e.g., Symbian, BB10, Harmattan



niche platforms

problems

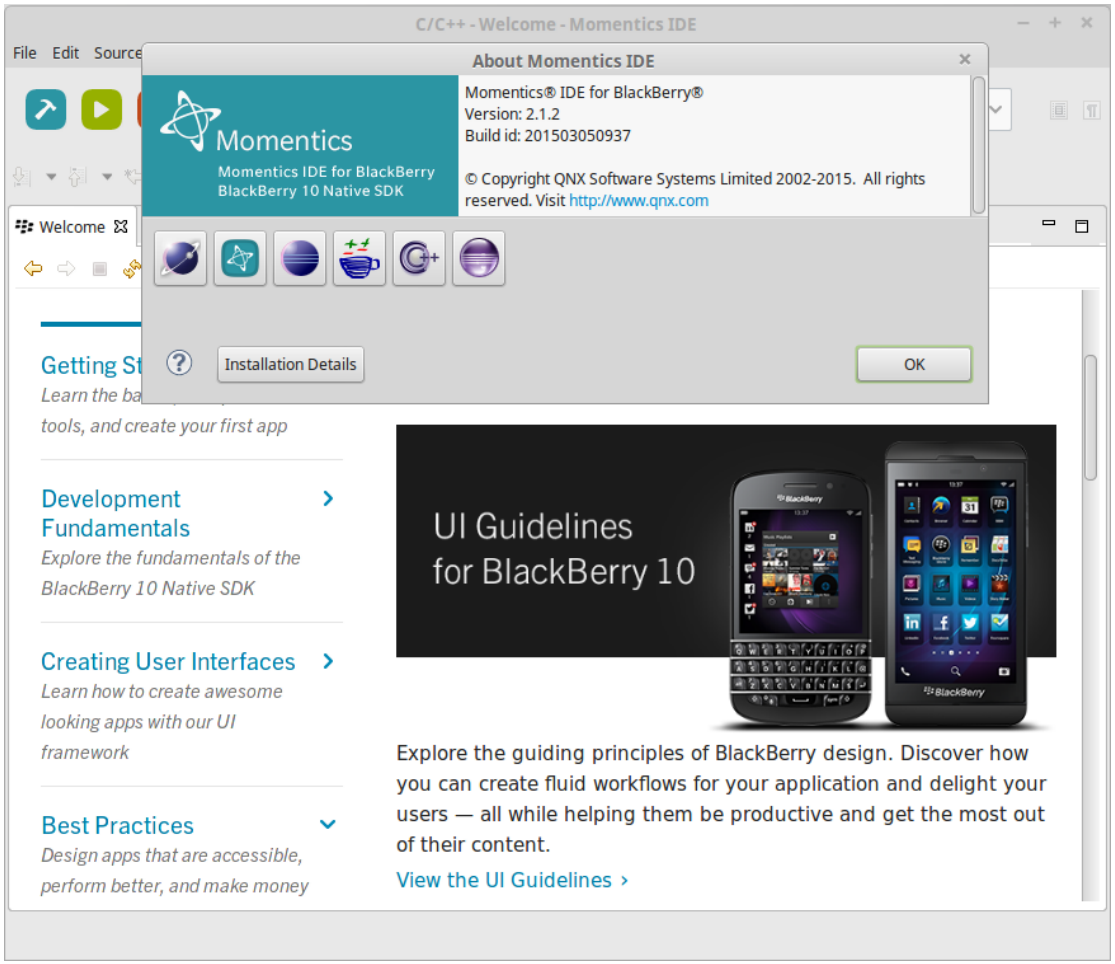
- few developers $\rightarrow$ socially limiting
- few libraries, tools $\rightarrow$ poor dev experience
- not established $\rightarrow$ discontinuation risk

advantages

- little competition $\rightarrow$ app discoverability, unit price, "bribery"

software development tools

- compiler
- IDE
- emulator
- on-target debugger
- build manager
- toolchain
- ...



... tools for packaging, installation, localization, file formats (executables, resource files, help files, bitmaps, certificates, printer drivers, ...), ...

unfamiliar tools

Symbian toolchain

`abld`, `bldmake`, `bmconv`, `elf2e32`, `makedef`, `makmake`, `rcomp`, …

languages

programming languages

- C, C++, JavaScript, ...

little languages

- project description languages
  - `qmake`, MMP, "tizen-manifest.xml", ...

custom and standard languages

makmake project (MMP)

```
TARGETTYPE exe
TARGET bldl_anyxporter.exe
UID 0x100039ce 0xeb1d1001
EPOCSTACKSIZE 0x10000
EPOCHEAPSIZE 0x020000 0x800000
CAPABILITY ReadUserData
```

"Symbian C++"

```
_LIT(KConsoleTitle, "Anyxporter");
CConsoleBase* console =
  Console::NewL(KConsoleTitle,
    TSize(KConsFullScreen, KConsFullScreen));
CleanupStack::PushL(console);
```

APIs / vocabulary

Symbian

`CBase`, `CActive`, `User`, `CleanupStack`, ...

Qt

`QObject`, `QString`, `QList`, `QMap`, `QVariant`, ...

Tizen 2.3

`tizen_error_e`, `event_cb`, `event_handler_h`, ...

platform lifespans

Psion Series 5

- 1997–2001
- EPOC release 5
  - became Symbian OS

Pebble

- 2013–2016
- Pebble OS
  - at Fitbit?

"Burning Platform" by Micky Aldridge (CC BY 2.0)

part 2: strategy

- **Core asset management:** Manage software components, target-specific build information, and program configurations in an open-world and tool-assisted manner.
- **Production tooling:** Have tooling complement and integrate with a programming language family sharing a common style and development environment. Accommodate target-platform-specific tools, but avoid depending on them.
- **Program composition:** Have at least one of the language family members include a component system supporting parameterizable components, and separate abstract "concepts" that may specify both syntax and semantics (as signatures and algebraic laws) to constrain composition. Have the language possess a static reasoning and translation friendly

constraint

- play nice with platforms (APIs, languages, tools)
  - coexist, do not fight

product families

- multiple product configurations
  - to suit different platforms

product lines

- *systematically manage* multiple product configurations
  - to suit different platforms


wanted: "containment"

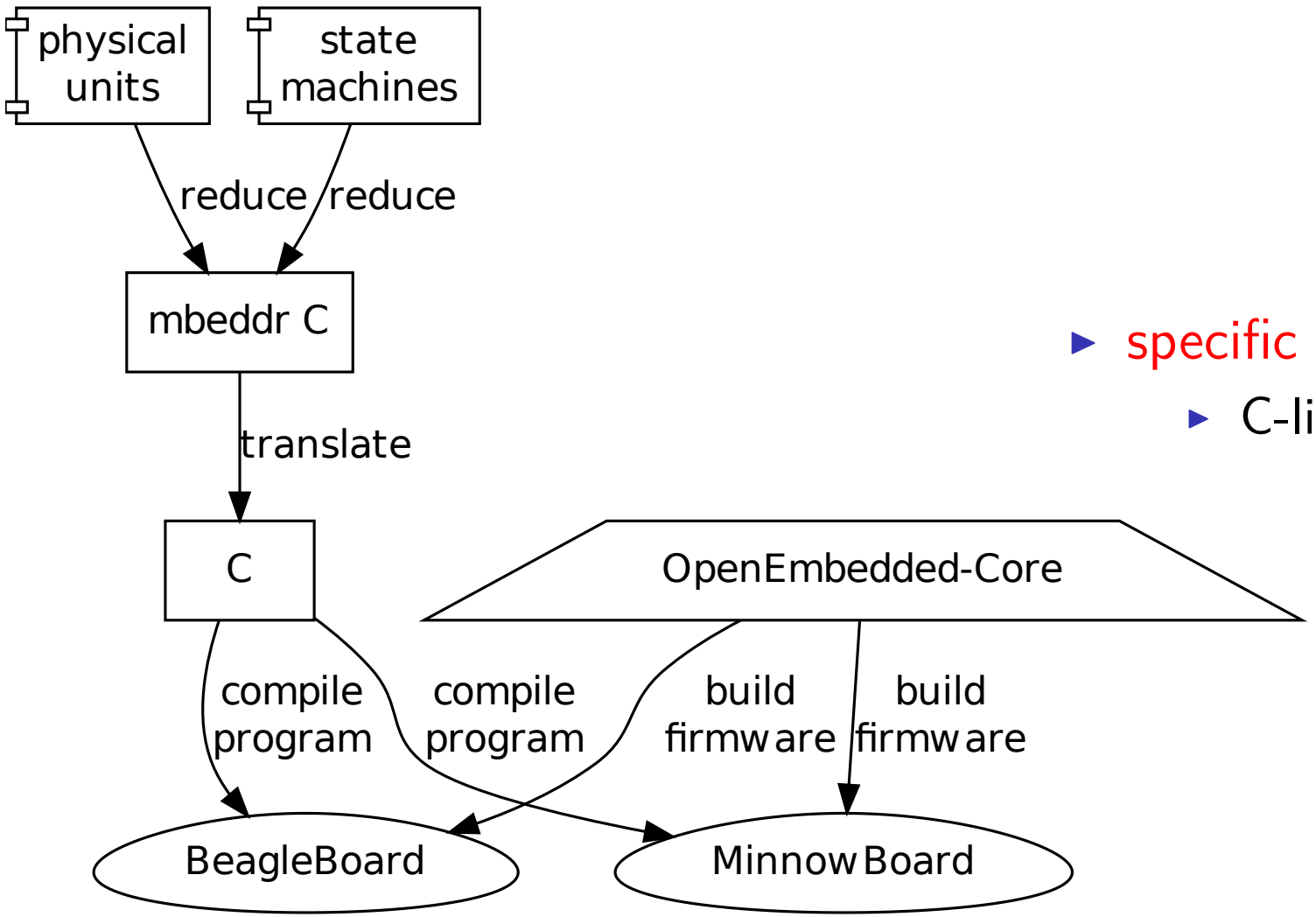
- knowledge about platform
  1. languages
  2. APIs
  3. tools

$\rightarrow$ limit the extent to which one must acquire and remember it

managing platform specifics: languages

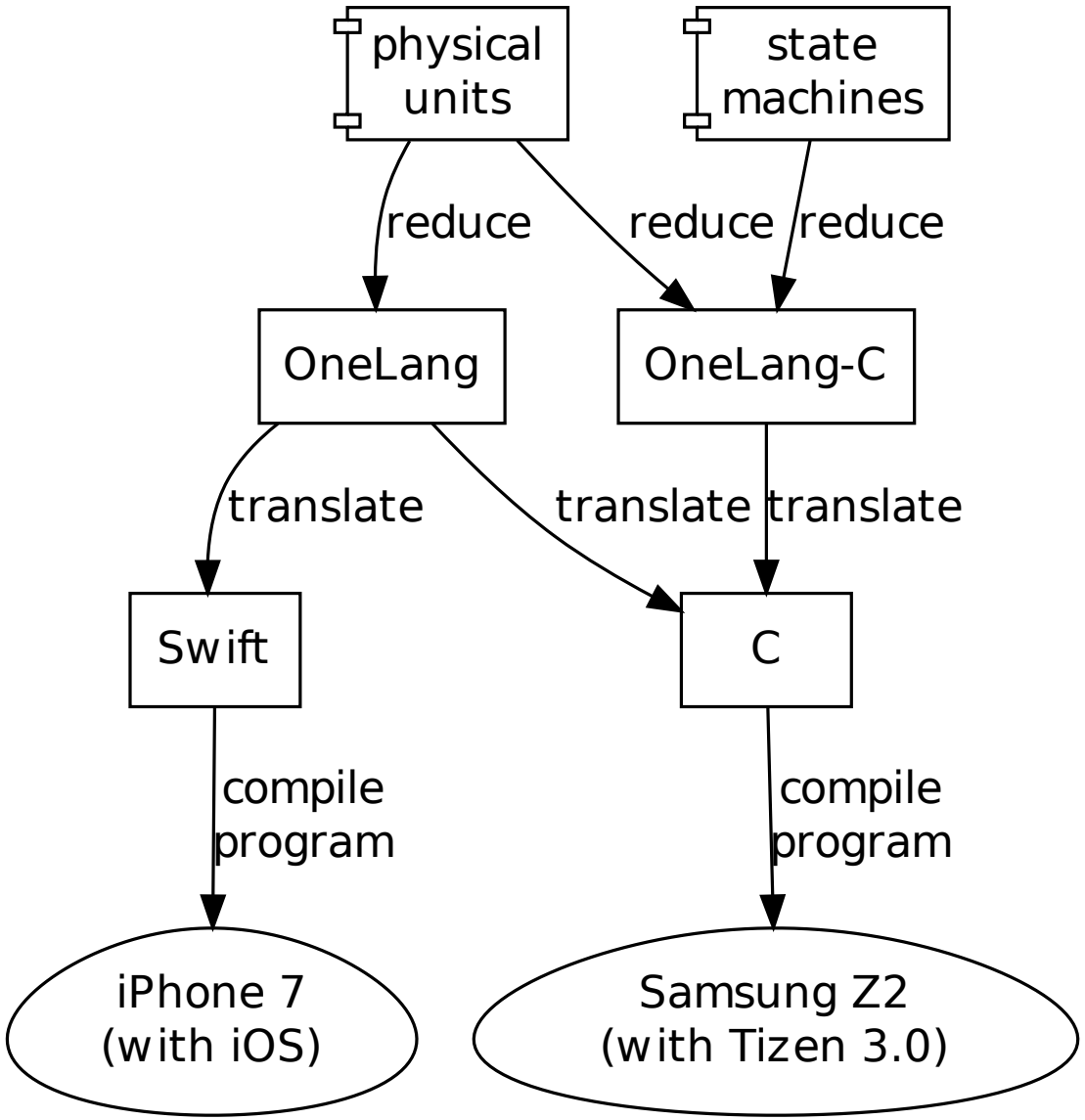
- program in a *familiar* translatable language
  - one to shield us from target languages
  - one that we control
    - can customize to capture idioms, etc.

single target language scenario



- specific mbeddr C
  - C-like language

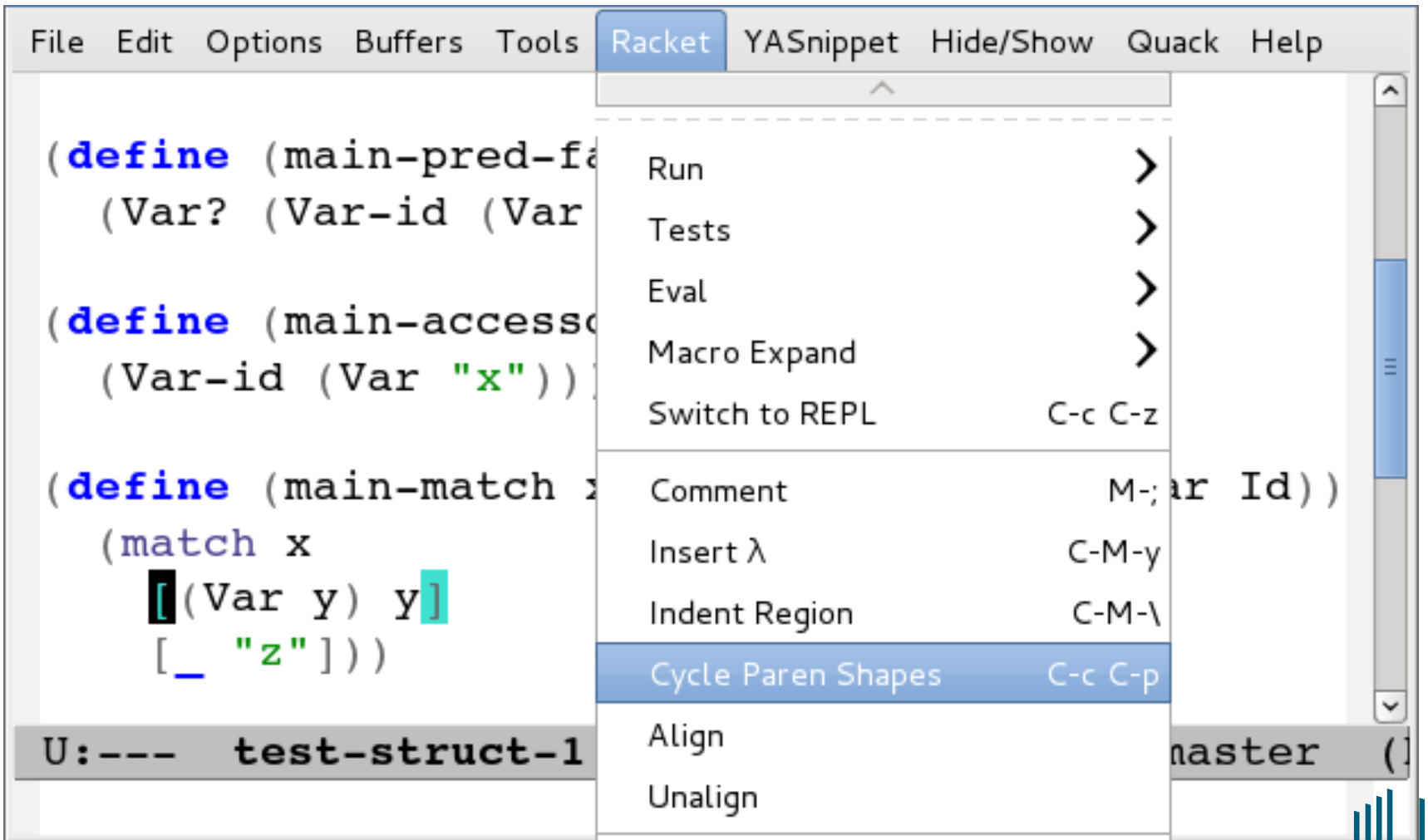
multiple target language scenario



- agnostic OneLang
  - unoriented language
- specific OneLang_C
  - C-oriented language

one language technology base

- "one" language to rule all them platforms—through variation
  - same look and feel
  - same programming environment



managing platform specifics: APIs

- hide specifics under abstract APIs

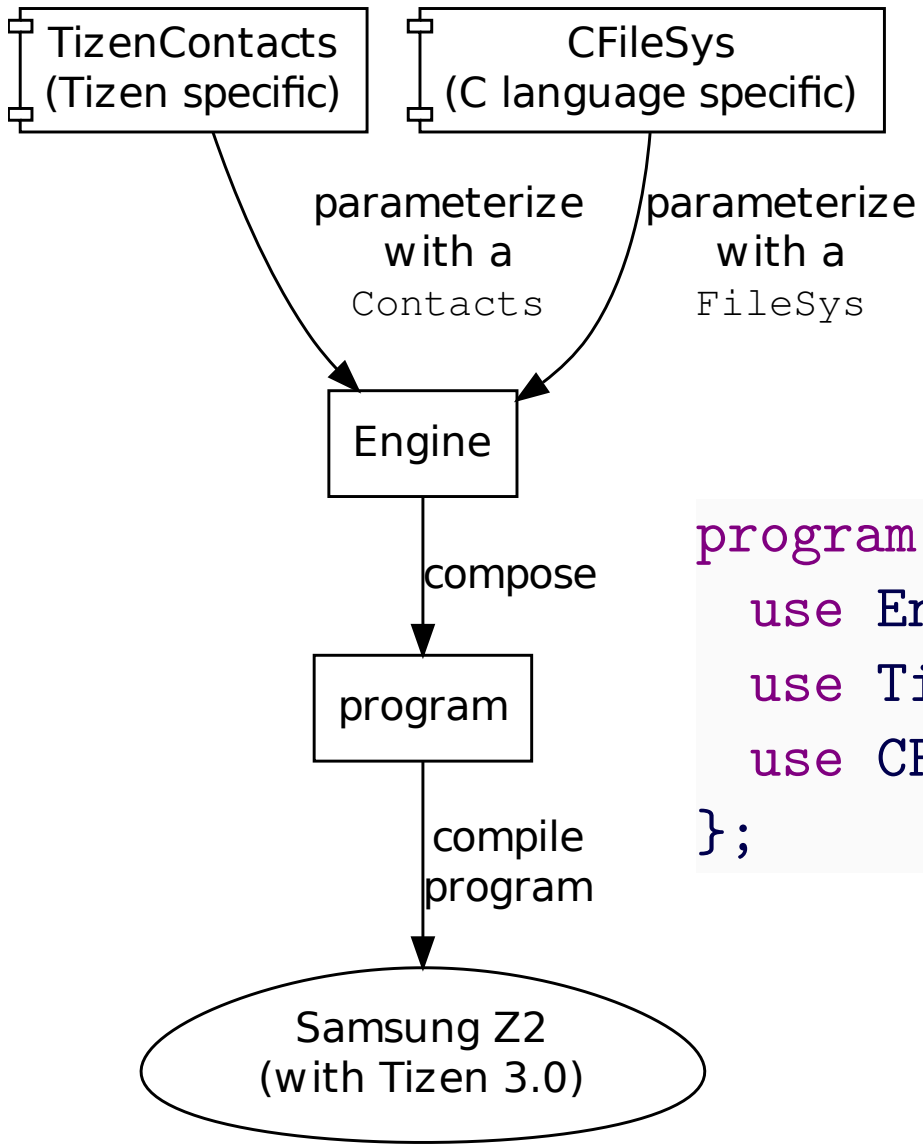
```
typedef struct _Engine* Engine; // abstract data type
Engine Engine_new(MyError* error);
boolean Engine_export_all_contacts(
  Engine obj, const char* filename, MyError* error);
void Engine_destroy(Engine obj);
```

- wanted: API parameterization for purposes of code composition
  - e.g., code to use for reading contacts
  - e.g., code to use for writing to a file

agnostic API management language

- e.g., Magnolia 

program composition




```
program TizenContactsExporter = {
  use Engine;
  use TizenContacts;
  use CFileSys;
};
```

managing platform specifics: tools

- write makefiles or scripts to drive vendor tools
  - use tools to source build configuration information
    - e.g., from API annotations (§4)

```
PLATFORM := symbian
PERMISSIONS := NetworkServices ReadUserData
```


part 3: technologies

Source-to-Source Compilation via Submodules

Tero Hasu
BLDL and University of Bergen
tero@ii.uib.no

Matthew Flatt
PLT and University of Utah
mflatt@cs.utah.edu

Managing Language Variability in Source-to-Source Compilers by Transforming Illusionary Syntax

Tero Hasu*

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
tero@ii.uib.no

Inferring Required Permissions for Statically Composed Programs

Tero Hasu, Anya Helene Bagge, and Magne Haveraaen

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
http://www.ii.uib.no/~{tero,anya,magne}

Errors as Data Values

Tero Hasu Magne Haveraaen
Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
http://www.ii.uib.no/~{tero,magne}

§2–5 technologies

- for adaptable, translatable programming languages

§2–3 language processing

- for implementing adaptable, translatable programming languages

Source-to-Source Compilation via Submodules

Tero Hasu
BLDL and University of Bergen
tero@ii.uib.no

Matthew Flatt
PLT and University of Utah
mflatt@cs.utah.edu

- European Lisp Symposium (ELS 2016)
  - Kraków

§2 Source-to-Source Compilation via Submodules

presents

A technique for arranging for further compilation of Racket languages, post macroexpansion and other desired processing.

achieves

- allows extensive reuse of Racket mechanisms
- retains support for separate compilation

§2 Magnolisp

- proof-of-concept software
- transcompiled, with a C++ back end

```
#lang magnolisp

(typedef int
  #:: (foreign))

(define (f1 x)
  #:: (export
       ^(-> int int))
  (define (g) x)
  (g))
```

```
MGL_PROTO int f1_g( int const& x );

MGL_API_FUNC int f1( int const& x ) {
  return f1_g(x);
}

MGL_FUNC int f1_g( int const& x ) {
  return x;
}
```

§2 defining languages in Racket

- a `#lang` is implemented as a module
- specifies a "reader" to turn text into syntax objects
- exports variables, macros, core forms

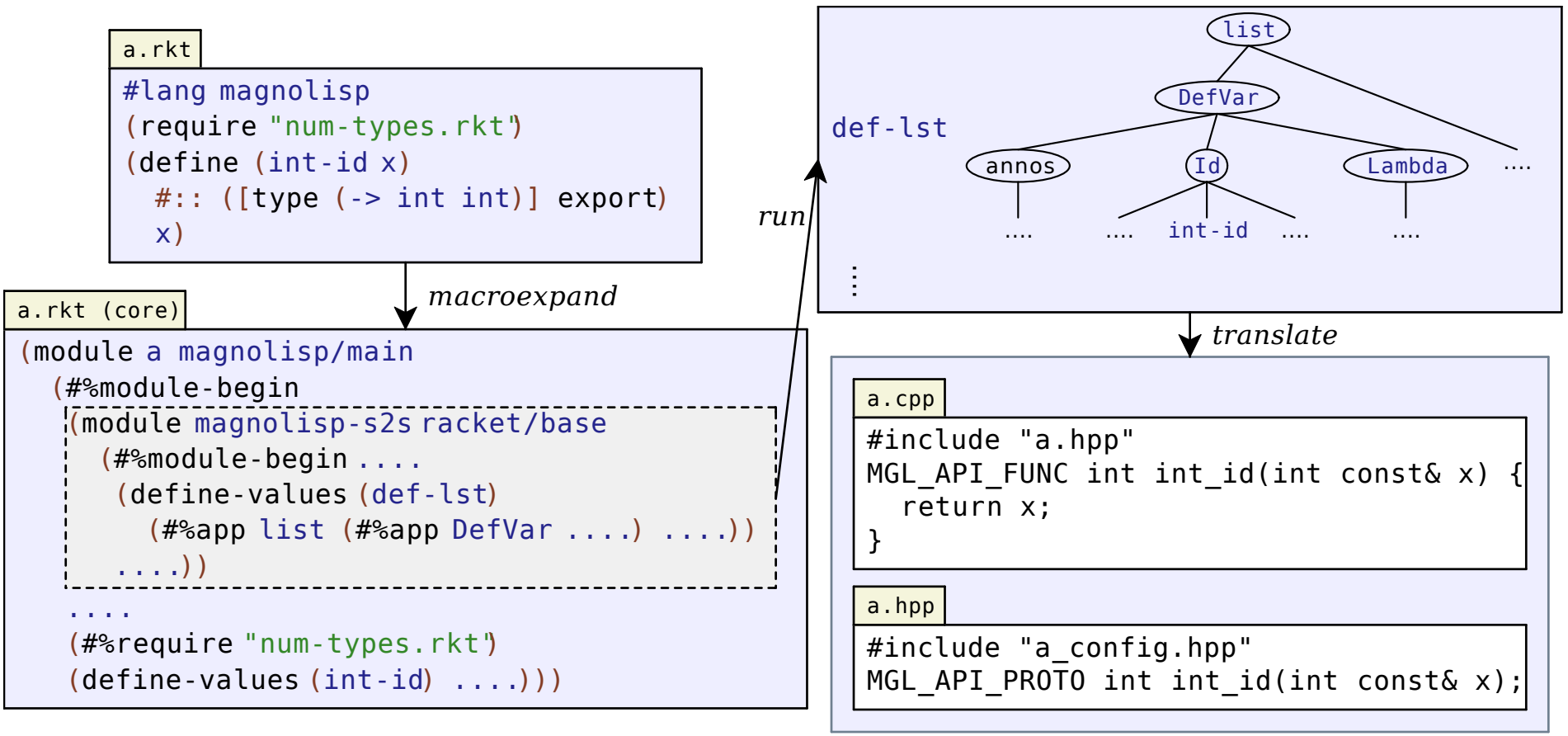
`another-magnolisp`

- just like `magnolisp`

```
#lang racket/base
(module reader syntax/module-reader
  another-magnolisp/main
  #:wrapper1 (lambda (t) (with-magnolisp-readtable (t)))
  (require magnolisp/reader-ext))
(require magnolisp)
(provide (all-from-out magnolisp))
```

§2 technique for source-to-source compilation

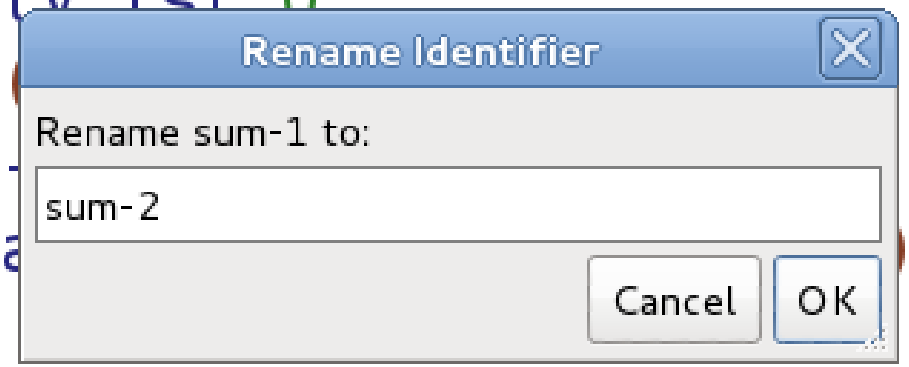
- allow a language's macros to target *foreign* core forms



§2 host language reuse

- reuse Racket's `#lang` mechanism for defining languages
- reuse Racket's language environment

```
(define (sum-1 lst) #:: (export)
  (if-empty lst 0
    (let (
      (if-
        (a                    )))))
```



- expose Racket's module system to your language
- expose Racket's macro system to your language
  - rarity: a "language workbench" for self-extensible languages

```
#lang magnolisp
(define-syntax-rule (if-not c t e)
  (if c e t))
```

§2 separate compilation

- macroexpand and byte-compile only out-of-date modules
  - e.g., with `raco make`

compatible with host language philosophy

- submodules *are* intended for defining new "phases"
  - here: transcompile time

alternative approaches

- e.g., expand externally and serialize into a separate file
  - more to manage yourself
  - still byte-compile modules for macroexpansion time use


§3

Managing Language Variability in Source-to-Source Compilers by Transforming Illusionary Syntax

Tero Hasu*

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
tero@ii.uib.no

- International Workshop on Open and Original Problems in Software Language Engineering (OOPSLE 2014)
  - Antwerp

more recently

- joint work with Anya Helene Bagge

§3 Illusionary Abstract Syntax

presents

A scheme for declaratively (in an embedded DSL) implementing more-abstract-than-usual abstract syntax tree data types.

achieves

- ASTs with abstract data types
- with some extra flexibility for commonality expression
  - potential for further DSL innovations
- expects compile-time expressive power, little run-time

§3 general idea

Use a macro-based embedded DSL for declaring

- actual data representations; and
- illusionary ones over the above.

§3 node data type (NDT)

- with named fields
- unrelated to other (host language) data types
- treated as abstract data—opaque, with operations
  - predicate, field access, construction
- patterns defined, for matching
  - as (special) macros
  - translating to operation uses


§3 view data type (VDT)

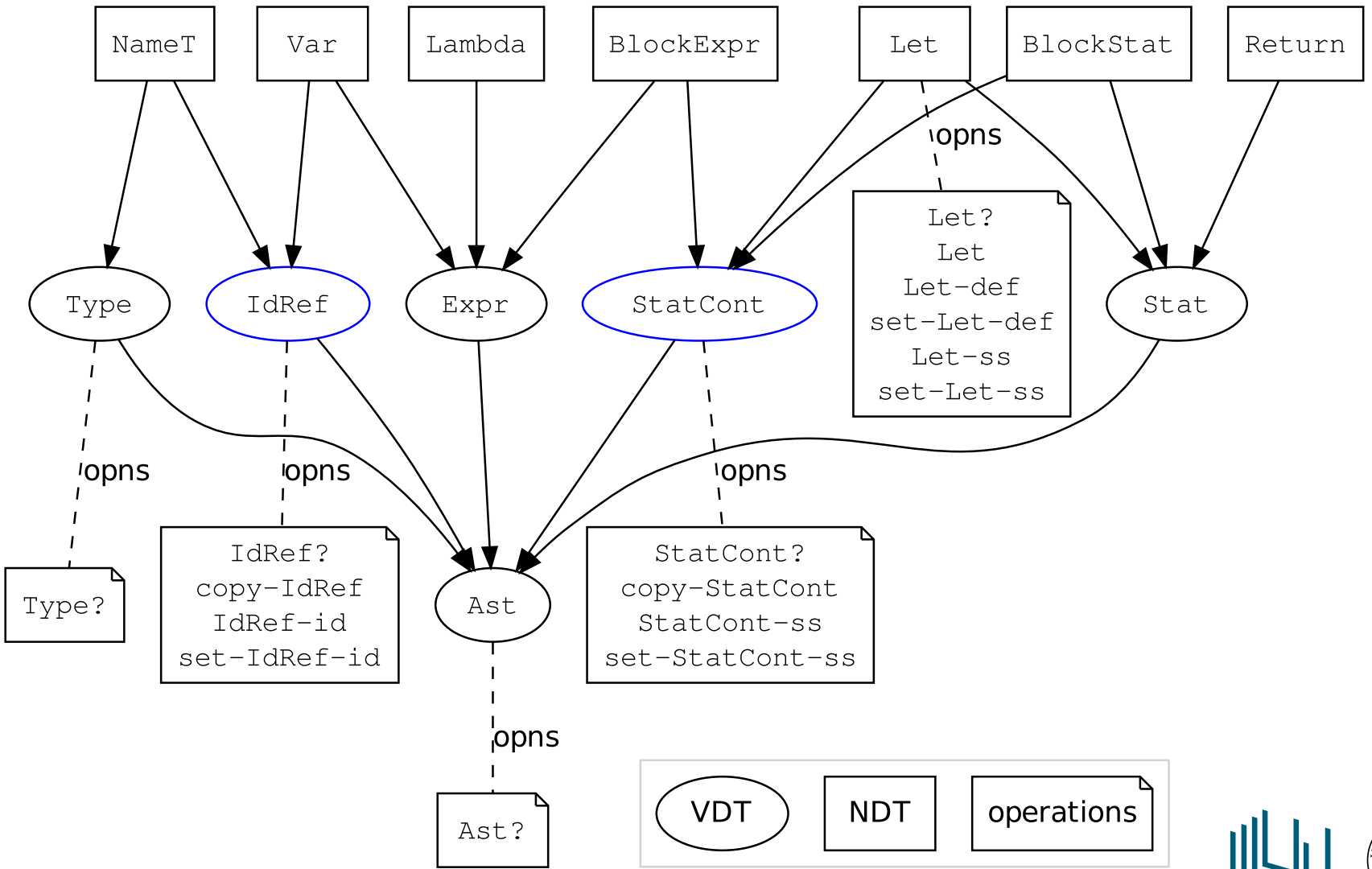
- with named fields
- uses other type(s) for storage
- treated as abstract data—opaque, with operations
  - predicate, field access, copying
- patterns defined, for matching
  - as (special) macros
  - translating to operation uses


§3 Illusyn

- a library for Racket
- used in Magnolisp implementation
- includes Stratego-style HoFs for rewriting strategies

§3 VDTs relate NDTs

- can also relate subsets of NDTs



§3 VDT vs. NDT APIs

VDT

```
(define-view V
  ([#:field v #:use n]))
(V? (N 0)) ;;=> #t
;; N/A
(copy-V (N 2) 0) ;;=> (N 0)
(V-v (N 3)) ;;=> 3
(set-V-v (N 4) 0) ;;=> (N 0)
(match (N 5) [(V v) v]) ;;=> 5
(V=? (N 6) (N 6)) ;;=> #t
```

NDT

```
(define-ast N (V)
  ([#:none n]))
(N? (N 0)) ;;=> #t
(N 1) ;;=> (N 1)
(copy-N (N 2) 0) ;;=> (N 0)
(N-n (N 3)) ;;=> 3
(set-N-n (N 4) 0) ;;=> (N 0)
(match (N 5) [(N n) n]) ;;=> 5
(N=? (N 6) (N 6)) ;;=> #t
```

- enumerating substructure needs disambiguation
  - view-directed traversals supported by Illusyn

§3 product-line use

- language translation through successive rewrites
- goal: more general and reusable transformation routines
  - for sharing among core language processors

§4–5 language features

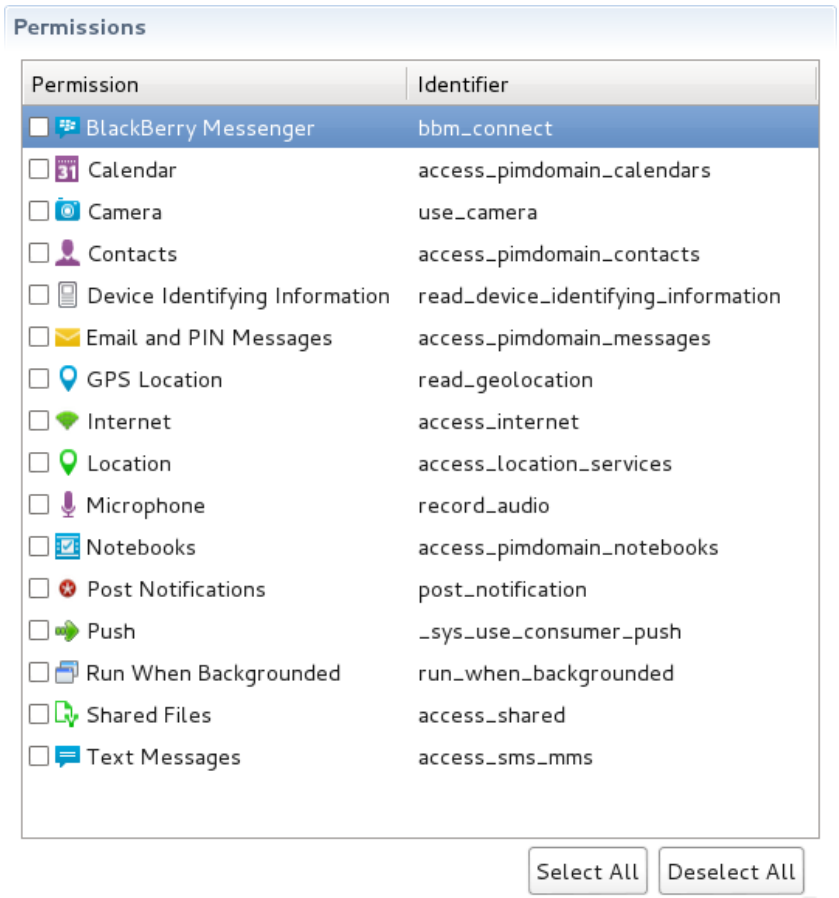
- for inclusion in agnostic, translatable programming languages
  - error prevention
  - error handling

§4 permissions

- permission-based security models
  - popularized by smartphone OSes
  - access control for sensitive APIs

permission requirements

- tend to be vendor specific
- can vary even between the releases of a single platform
- for a developer to declare for programs
  - optimal set, ideally



Inferring Required Permissions for Statically Composed Programs

Tero Hasu, Anya Helene Bagge, and Magne Haveraaen

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
http://www.ii.uib.no/~{tero,anya,magne}

- 18th Nordic Conference on Secure IT Systems (NordSec 2013)
  - Ilulissat

§4 Inferring Required Permissions for Statically Composed Programs

presents

A solution for cross-platform permission management.

achieves

- tool support for inferring platform-specific permission requirements from code
- language support for abstracting over run-time permission errors so that they can be handled platform agnostically

§4 proof-of-concept implementation

Magnolia

- permission inference (was) integrated into its implementation

Anyxporter

- example app (available)
- `https://github.com/bldl/anyxporter`
  - `magnolia` branch

§4 desirable language characteristics

- interface-based abstraction
  - to support organizing cross-platform codebases
- programs are amenable to extensive and accurate reasoning
  - e.g., by restricting language
  - e.g., by allowing declaration of properties


§4 agnostic error reporting abstraction: alerts

Stayin' Alert: Moulding Failure and Exceptions to Your Needs

Anya Helene Bagge Valentin David Magne Haveraaen Karl Trygve Kalleberg

University of Bergen, Norway

{anya,valentin,magne,karltk}@ii.uib.no

Abstract

Dealing with failure and exceptional situations is an important but tricky part of programming, especially when reusing existing components. Traditionally, it has been up to the designer of a library to decide whether to use a language's exception mechanism, return values, or other ways to indicate exceptional circumstances. The library user has been bound by this choice, even though it may be inconvenient for a particular use. Furthermore, normal program code is often cluttered with code dealing with exceptional circumstances.

This paper introduces an alert concept which gives a uniform interface to all failure mechanisms. It separates the handling of an exceptional situation from reporting it, and allows for retro-fitting this for existing libraries. For instance, we may easily declare the

of SPARK Ada [2], where the Ada exception mechanism has been removed in an attempt at making validation and verification easier. This ideal advocated by such a "keep errors out" approach is certainly desirable. It is generally preferable to write algorithms with as few corner cases as possible.

In many cases, however, removing the errors altogether is simply not feasible [27]. Most modern applications run in multi-user, multi-process environments where they share resources such as storage and network with other applications. In these situations, operations on files, network connections and similar operating system resources can always fail, due to interaction with other programs on the running system or external devices.

Errors and exceptional situations need not always be caused by external factors, however. Even in situations where resource re-

§4 solution: dynamic behavior

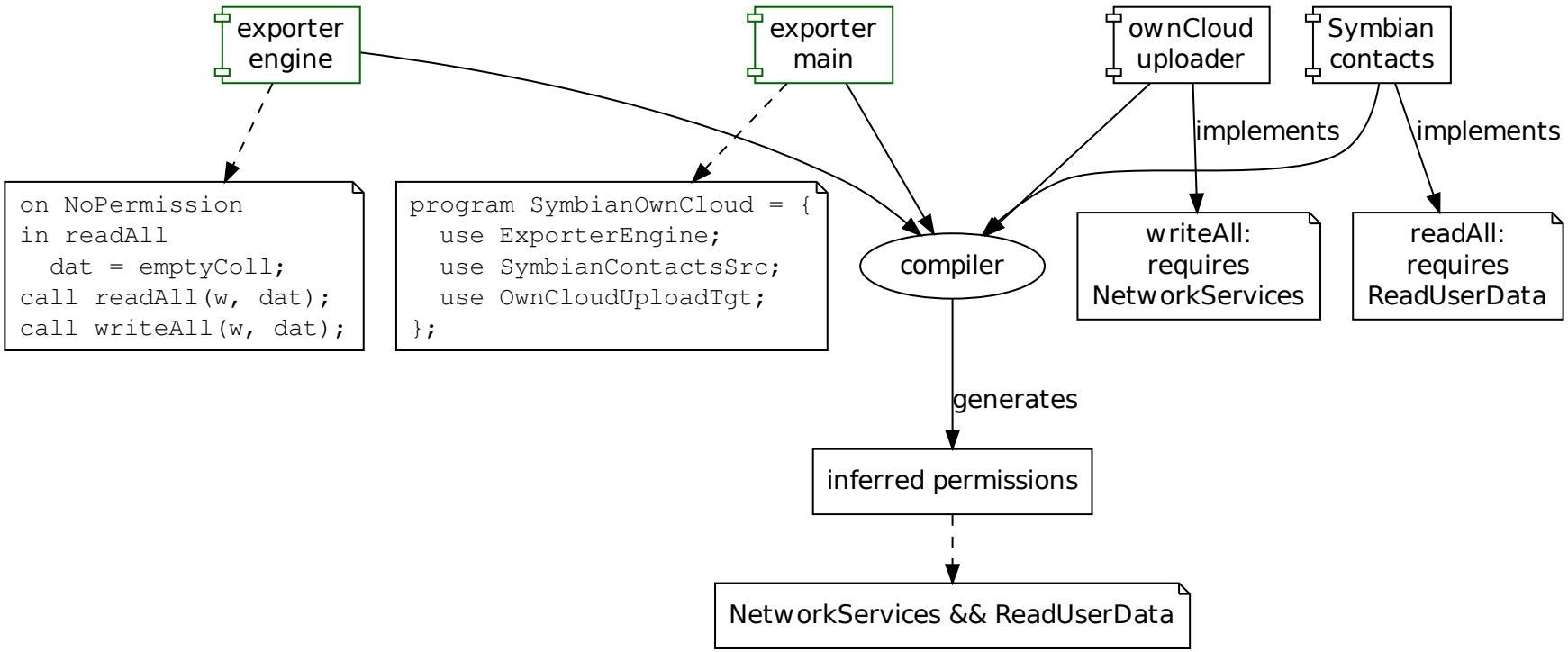
1. declare possible run-time permission errors agnostically
   - e.g., `E_PRIVILEGE_DENIED` return value
     $\mapsto$ `NoPermissionSocial` alert
     - example code in §5


§4 solution: static requirements

2. declare platform-specific permission requirements specifically
   - per operation, per implementation
     - if opaque (i.e., foreign)
   - as a predicate expression—commonly need $\wedge$, sometimes $\vee$
     - for tools to statically infer permission requirements for a program
     - e.g., `NetworkServices` $\wedge$ `ReadUserData`

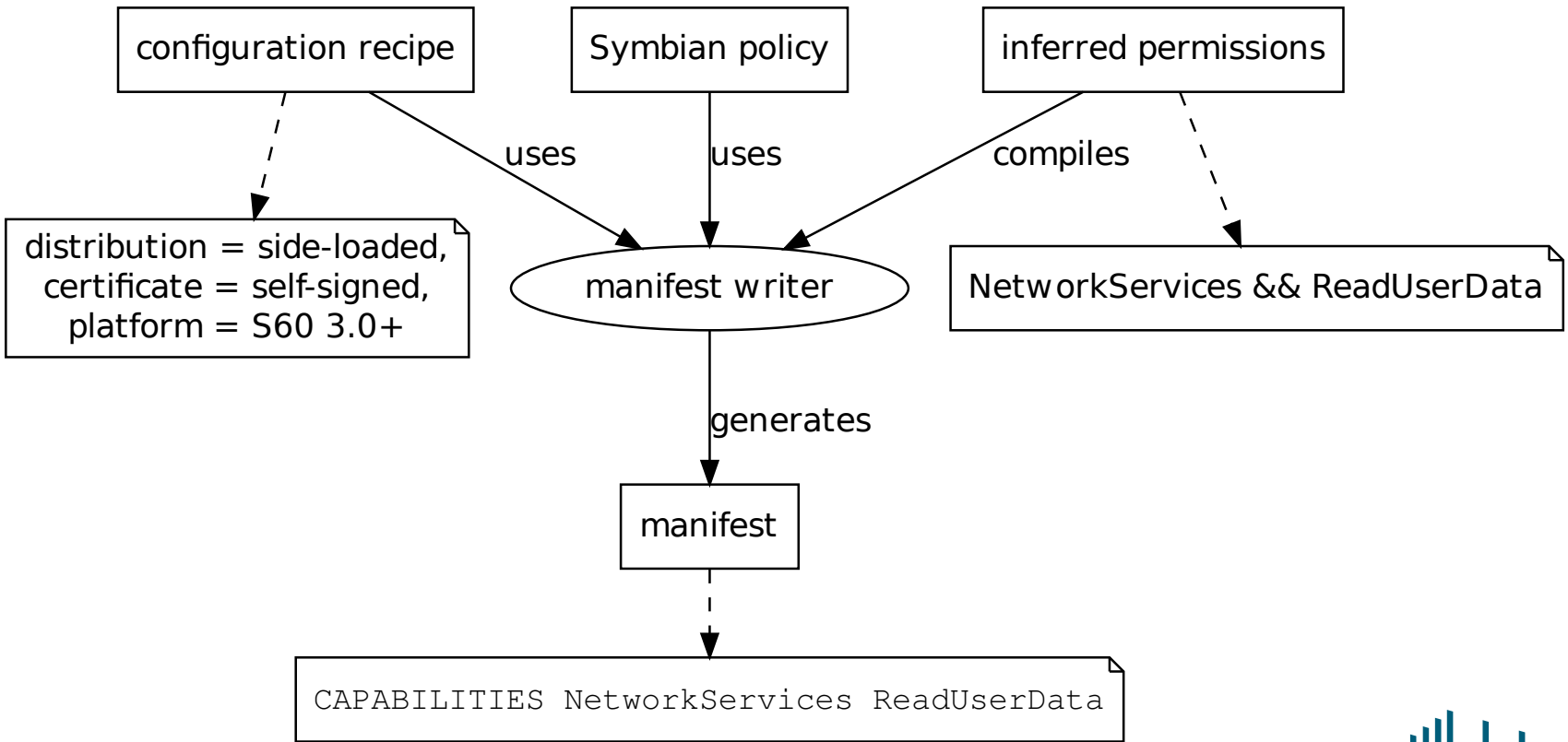

§4 solution: static analysis

3. infer programs' permission requirements
   - based on their reachable uses of operations



§4 solution: platform-specific policy

4. decide on permission requests, using
   - program configuration information
   - platform-specific policies
5. insert requests into vendor tools' "manifest" files



Errors as Data Values

Tero Hasu Magne Haveraaen
Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
http://www.ii.uib.no/~{tero,magne}

- Norwegian Informatics Conference (NIK 2016)
  - Bergen

§5 Declarative Propagation of Errors as Data Values

presents

Portable, non-disruptive, guarded-algebra-inspired error reporting convention, and language (wide) support for it.

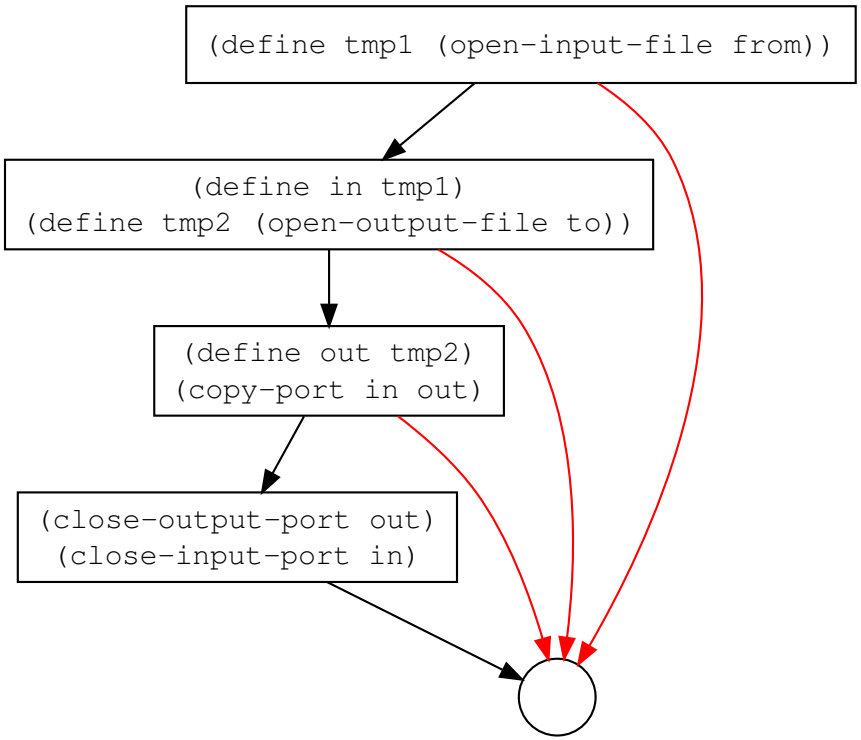
achieves

- allows referentially transparent expression language
- accommodates “normal” exception syntax

§5 *possibility* of exceptions vs. static reasoning

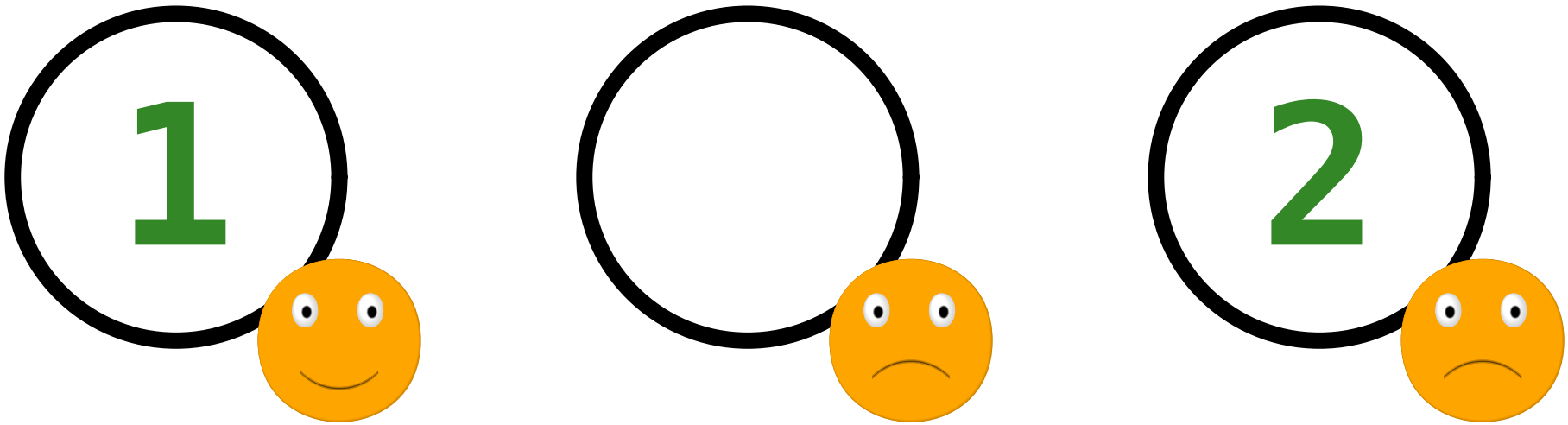
"Exceptions are not exceptional enough." (Liang et al.)

```
(define in
  (open-input-file from))
(define out
  (open-output-file to))
(copy-port in out)
(close-output-port out)
(close-input-port in)
```



§5 data and control flow

- shall we just have abnormal data instead of abnormal control?
  - keep on computing despite uncomputable or unacceptable values



Erda family of languages

- all language-native data values are either *good* or *bad*
- all operations appear total

§5 file copying in Erda$_{GA}$

```
(define in (open-input-file from))
(define out (open-output-file to))
(copy-port in out)
(close-output-port out)
(close-input-port in)
```

- no disruptive flow any longer
- now with safe resource cleanup
  - resource cleanup bookkeeping comes for “free”
  - but must not try calling primitives with invalid arguments


§5 guarded algebras

"error history extension" for primitive functions

$$\hat{f}(\hat{a}_1, \ldots, \hat{a}_k) = \begin{cases} \mathrm{cgood}(f(a_1, \ldots, a_k)) & \text{if } \hat{a}_1 = \mathrm{cgood}(a_1), \ldots, \\ & \hat{a}_k = \mathrm{cgood}(a_k) \text{ and} \\ & a_1, \ldots, a_k \text{ are} \\ & \text{good arguments for } f, \\ \mathrm{cbad}(`f(\hat{a}_1, \ldots, \hat{a}_k)) & \text{otherwise.} \end{cases}$$

§5 total language in Erda$_{GA}$

- Erdas extend the entire language, guarded algebra style
  - just operate—bad happenings become values

```
> (define bad (raise 'bad))
> bad
(Bad bad: raise bad)
> (if #t 'true 'untrue)
(Good 'true)
> (if bad 'true 'untrue)
(Bad bad-arg: if-then (Bad bad: raise bad) <fun> <fun>)
```

- history of failed expressions recorded
  - also: redo semantics

§5 adapting to foreign conventions

- declaratively
  - a way to capture knowledge about error behavior

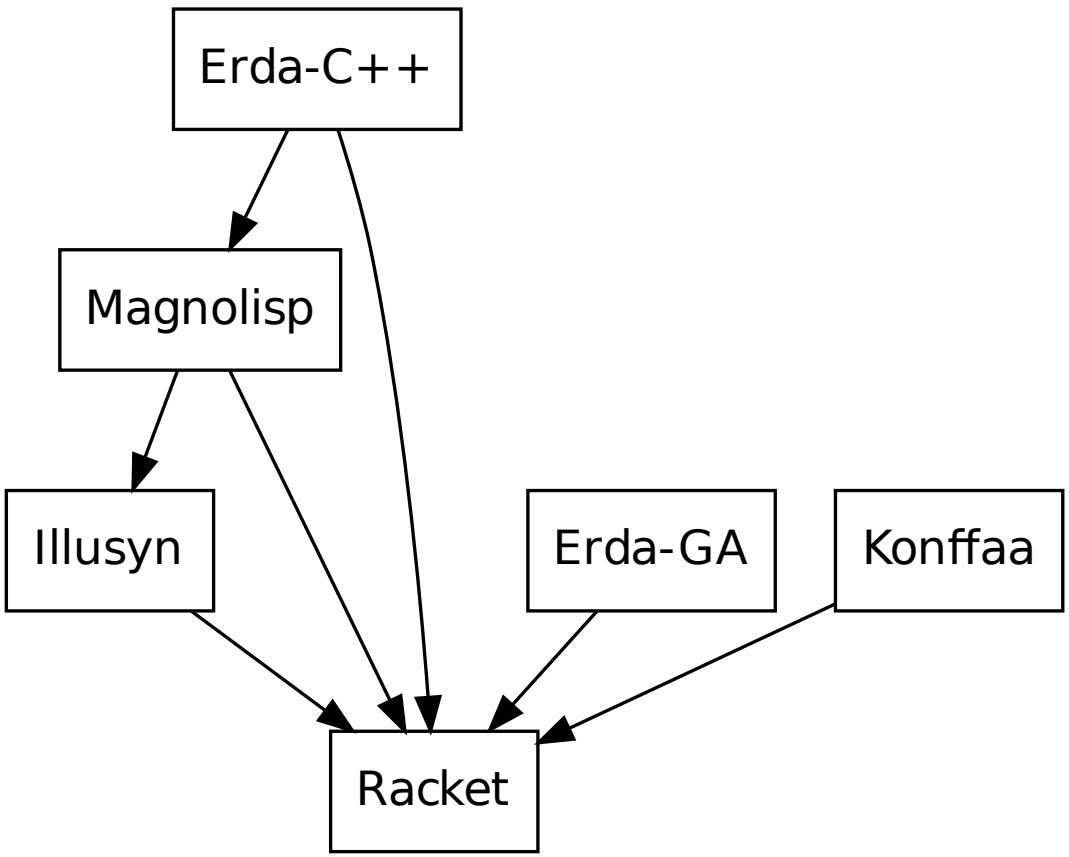
example in Erda$_{C++}$

- `E_PRIVILEGE_DENIED` return value
  $\mapsto$ `NoPermissionSocial` alert
  - example from §4

```
(declare (read-all-contacts db)
  #:: ([type (-> ContactsDatabase ContactsSet)])
  #:alert ([NoPermissionSocial post-when
            (= value E_PRIVILEGE_DENIED)]))
```


proof-of-concept software

- `https://bldl.ii.uib.no/software/pltnp/`
- Erdas, Illusyn, Konffaa, Magnolisp
  - Erda_{C++}, Erda_{GA}, ...



Erda_{C++}

- a Magnolisp-based language

summary

1. niche platforms
2. a strategy for dealing with them
3. technologies for that
   - source-to-source compiled Racket languages
   - AST abstract data types, declaratively
   - permission inference for composed programs
   - portable error handling, with local control flow

programming language technology

- §2 macro and module system reuse for translated languages
- §3 declared abstract data types for (more) abstract syntax
- §4 platform-agnostic permission management
- §5 portable and semi-declarative error handling

ultimate goal

Develop a strategy and agnostic-but-specializable languages and tools for targeting *any* platform. More platforms, more opportunities.

software and documentation

`https://bldl.ii.uib.no/software/pltnp/`

contact

tero@ii.uib.no