Banshee Tutorial and Overview

John Kodumal (University of California at Berkeley)
jkodumal@cs.berkeley.edu

March 29, 2006

Contents

1 Introduction

Banshee is a highly optimized toolkit for constructing constraint-based program analyses. Banshee, like its predecessor Bane [1], is based on *mixed constraints*, allowing users to design their own ad-hoc analysis formalisms by mixing standard, well understood constraint languages [3]. Banshee's main innovation is its use of an analysis specification language to *specialize* the constraint resolution engine for specific program analyses. This approach yields several distinct advantages over previous toolkits:

- **Cleaner user interfaces.** Given an analysis specification, Banshee creates a compact interface tailored to the analysis.
- **Better type safety.** Constraints are typically subject to a set of *well-formedness* conditions. In previous toolkits, these conditions were checked dynamically. In Banshee more of these conditions are checked statically, reducing the possibility of run-time errors.
- **Improved performance.** Banshee applications realize a performance gain as an effect of reducing these dynamic checks and generating specialized C code.
- **Better extensibility.** The system can easily be extended to handle new constraint formalisms.

Other novel features in Banshee include an efficient implementation of persistence (constraint systems can be quickly saved/loaded), and a backtracking capability (constraint systems can be rolled back to any previous state) that can be used for "poor-man's" incremental analysis [7].

There are a number of ways for an analysis designer to use Banshee. These will be explained in detail later, but we provide an overview here:

- **As an interpreter** (Section 4). Banshee comes with an interpreter, IBanshee `[bin/ibanshee.exe]`. IBanshee allows the designer to interact with Banshee by typing in constraints directly. This is a handy tool for learning about set constraints. It's also an easy way to interface with Banshee, especially if the client is written in a language other than C.
- **As a generic C library** (Section 5). Here, the analysis designer simply links their client to a C library `[engine/libnsengine.a]`. This library has a generic (nonspecialized) interface `[engine/nonspec.h]`. This approach forgoes the benefits of specialization, but allows new constructors to be defined dynamically.
- **As a specialized C library** (Section 6). Here, the analysis designer writes a short specification file describing the constraint language needed for her analysis. The Banshee code generator `[bin/banshee.exe]` is invoked on this file and produces a specialized constraint solver interface. The analysis client and the specialized interface are then linked with a backend C library `[engine/libengine.a]`.

- **As a context-free language reachability engine** (Section 7). BANSHEE also includes a wrapper API `[dyckcfl/dyckcfl.h]` that implements a very efficient reduction from a restricted class of context-free language reachability to set constraints. Many program analysis problems fall into this restricted class, so we have implemented an API to express these problems.

We are also working on foreign function interfaces for BANSHEE. These would simplify writing analyses in languages other than C. Currently, there is a Java interface (via JNI) for the context-free language reachability API. We plan to have Java and O'Caml interfaces for the entire BANSHEE system soon.

2 Installation

2.1 Prerequisites

Here is a list of the software packages required to build BANSHEE:

- Objective Caml version 3.0.8.2 or later (available at http://caml.inria.fr)
- GCC, the Gnu Compiler Collection, version 3.3 or later
- Python, version 2.3.3 or later
- indent
- etags
- flex
- bison

With the exception of Objective Caml, these utilities should be standard.

2.2 Compiling Banshee

Assuming all prerequisites are installed, you can compile banshee by typing `make all check` or `gmake all check` at the command line in the top level of the source tree. If everything succeeds, make will report `SUCCESS: All check targets passed.` and the following targets will be built:

- `[bin/banshee.exe]`, the BANSHEE code generator, and `[engine/libengine.a]`, the backend constraint library. These targets are required to use BANSHEE as a specialized engine.
- `[engine/libnsengine.a]`, the nonspecialized constraint library. This target is required to use BANSHEE as a generic constraint library or context-free language reachability engine.

- `[bin/ibanshee.exe]`. This target is IBANSHEE, the BANSHEE interpreter.
- `[dyckcfl/dyckcfl.o]`. This object file is required (in addition to `[engine/libnsengine.a]`) to use BANSHEE as a context-free language reachability engine.

`make` will also build the user manual (this document) a suite of small test applications (see `[tests/README]`) and the points-to analysis application (see `[cparser/README]`).

Here is a summary of the most useful `make` targets:

- `make all` will build everything.
- `make check` will verify your build.
- `make banshee` will build banshee without building any test applications.
- `make docs` will just build the documentation.
- `make points-to` will build the points-to analysis application and any dependencies.
- `make ibanshee` will build IBANSHEE and any dependencies.

BANSHEE should build and run on Linux, FreeBSD, Windows (with cygwin) and MacOS X. On FreeBSD, make sure to use `gmake` instead of `make`.

3 Tutorial

In this section, we give a brief tutorial and gentle introduction to set constraints. We'll explain the basic formalism and demonstrate how to express some simple problems in IBANSHEE. For those who have some familiarity with set constraints, this tutorial covers the simple term-set model of set constraints. The complete BANSHEE system supports a much richer model— we refer the interested reader to [3, 2] for further details.

3.1 Expressions

We'll begin our discussion with a few definitions. A *ranked alphabet* is a finite set of constructors and a function *arity*(...) that maps each constructor to a nonnegative integer (the *arity* of the constructor). We use $c, d, e, \ldots$ to range over constructors. 0-ary constructors are called *constants*. We also assume that there is a special set of constants called *variables* which are disjoint from our ranked alphabet. We'll use $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \ldots$ to range over variables.

Think of a ranked alphabet as a set of building blocks for making *expressions* (or trees).

Example 1 Suppose our alphabet is the set of constructors:

$$f, g, c$$

with $arity(f) = 2$, $arity(g) = 1$, $arity(c) = 0$. This alphabet defines a set of expressions over f, g, c, which we can build in the natural way. Examples of expressions are $f(\mathcal{X}, \mathcal{Y})$, $f(g(c()), c())$, $f(c(), c())$, $g(f(c(), c()))$. The last three expressions are *ground terms*, which are expressions that don't contain any variables. As shorthand, we'll often use c instead of $c()$ for constants.

More formally, the set of expressions over a ranked alphabet and set of variables is defined inductively:

- $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \ldots$ are all expressions
- c is an expression if $arity(c) = 0$
- $f(e_1, \ldots, e_n)$ is an expression if $arity(f) = n$ and $e_1, \ldots, e_n$ are expressions

3.2 Expressions in IBanshee

Before proceeding further, we'll pause and explain how to declare constructors and variables in IBanshee. Both constructors and variables must be declared before use. IBanshee constructors must begin with an upper or lower case letter, followed by a string of letters, numbers, or underscores. Declaring a constant is simple [1]:

```
[0] > c : setIF
constructor: c
```

Here, we've created a constant c. The colon followed by `setIF` is a *sort* declaration in IBanshee. Think of a sort as a type (like int or float in a conventional programming language). Since we're focusing on set constraints for now, we'll defer our explanation the other sorts that Banshee provides. For now, everything we make will be of sort `setIF`.

Declaring an n-ary constructor is slightly more involved, because we need to tell IBanshee the constructor's arity. We do so with a comma separated list of sort declarations:

```
[0] > f(+setIF,+setIF) : (setIF)
constructor: f
```

This declares a binary constructor f. Notice the plus signs before the sort declarations in this example. These are *variance* declarations. We'll defer discussion of variance declarations until later. For now, we'll just restrict ourselves to *positive* variances, which are specified with `+`.

[1] The `[0] >` is IBanshee's prompt. The line immediately following a line starting with a prompt is IBanshee's output. If you want to try these examples out, just type the remainder of each line after the prompt.

Variables are declared just as constants. However, IBanshee syntactically distinguishes variables from constructors by forcing you to begin each variable name with a tick (`'`). So for example:

```
[0] > 'x : setIF
var: 'x
```

declares a variable called `'x`.

Note that IBanshee only asks you to declare constructors and variables. There is no need to declare other terms explicitly. In other words, once you've defined `f` and `'x` as in the preceeding examples, you can refer to an expression like `f('x,'x)` without declaring it first. Of course, we have yet to explain how to do anything useful with expressions... so without further ado, let's talk about constraints!

3.3 Constraints

Constraints are inclusion relations between expressions. A constraint is a relation of the form $e_1 \subseteq e_2$, where e_1 and e_2 are expressions. A system of constraints is a finite conjunction of constraints. A solution S of a system of constraints is a mapping from variables to sets of ground terms such that all inclusion relations are satisfied. Let's see an example:

Example 2 Suppose that we have f, g, c as constructors with $arity(f) = 2$, $arity(g) = 1$, and $arity(c) = 0$ as before, along with variables $\mathcal{X}$ and $\mathcal{Y}$. Now consider the following system of constraints:

$$f(\mathcal{X}, g(\mathcal{X})) \subseteq f(\mathcal{Y}, \mathcal{Y})$$

$$c \subseteq \mathcal{X}$$

A solution to this system of constraints is S where $S(\mathcal{X}) = c$, $S(\mathcal{Y}) = c, g(c)$. You can see that this solution satisfies the inclusion constraints by substituting in for the variables.

A system of constraints may have no solutions, one solution, or many solutions. Often we will want the least solution or the greatest solution. For the simple model of constraints explained here, the least and greatest solutions will be unique (assuming there are any solutions at all!). In the previous example, the solution S is least.

3.4 Constraints in IBanshee

Creating a system of constraints in IBanshee is straightforward. After defining a set of constructors and variables, you can build expressions over those constructors and variables and add constraints. We'll continue with an IBanshee program that corresponds to the previous example:

```
[0] > f(+setIF,+setIF):setIF
constructor: f
[0] > c : setIF
constructor: c
[0] > g(+setIF) : setIF
constructor: g
[0] > 'x : setIF
var: 'x
[0] > 'y : setIF
var: 'y
[0] > f('x,g('x)) <= f('y,'y)
[1] > c <= 'x
```

Now that we have defined our constraint system, we'll probably want to inspect the solutions of the constraints. We can do so using IBanshee's query commands. IBanshee commands always begin with `!`. The first command is `!tlb`, which stands for *transitive lower bounds*. This command allows you to read off the least solution of the constraints:

```
[2] > !tlb 'x
{c}
[2] > !tlb 'y
{c, g('x)}
```

With a little investigation we see that this is exactly the least solution. But why doesn't IBanshee explicitly compute it? In other words, why doesn't IBanshee report that `'y` maps to the ground set {`c,g(c)`}? Another example is in order.

Example 3 Consider the following constraint system:

$$g(\mathcal{X}) \subseteq \mathcal{X}$$

$$c \subseteq \mathcal{X}$$

There is a solution set for $\mathcal{X}$, namely $\mathcal{X} = \{c, g(c), g(g(c)), \ldots\}$. However, there are no finite solutions for $\mathcal{X}$.

Thus, it may not be possible for Banshee to explicitly enumerate a set of ground terms as a solution for a given variable. A little more machinery is required to understand solutions and fully explain what the `!tlb` command computes. Briefly, it turns out that each set variable describes a *regular tree language* and that solutions of set constraints can be viewed as collections of regular tree grammars. The solution for $\mathcal{X}$ in the previous example corresponds to the regular tree grammar

$$X \Rightarrow c$$

$$X \Rightarrow g(X)$$

and we see that the `!tlb` command essentially returns the right-hand sides of the grammar productions corresponding to the solution for a given variable. A complete discussion is beyond the scope of this tutorial; we refer the interested reader to [4] for more details.

3.5 Variance

Recall the notation we used to define the binary constructor `f` in IBANSHEE:

```
[0] > f(+setIF,+setIF):setIF
constructor: fun
```

Let's explain the meaning of the `+` syntax. We'll start with a simple example:

Example 4 Suppose we have a binary constructor *pair* that models sets of pairs. Consider the sets $\mathcal{X} = \{1, 2\}$ and $\mathcal{Y} = \{3\}$. Then $pair(\mathcal{X}, \mathcal{Y}) = \{pair(1, 3), pair(2, 3)\}$. Now, suppose we add 4 to the set $\mathcal{Y}$. Intuitively, $pair(\mathcal{X}, \mathcal{Y})$ should grow to $\{pair(1, 3), pair(2, 3), pair(1, 4), pair(2, 4)\}$, so $pair(\mathcal{X}, \mathcal{Y})$ grows as $\mathcal{Y}$ grows. Notice that $pair(\mathcal{X}, \mathcal{Y})$ grows as $\mathcal{X}$ grows, as well. In general, the set $pair(e_1, e_2)$ grows if either of e_1 or e_2 grows.

We say that a constructor c has *positive* variance in its ith position if the set $c(\ldots, e_i, \ldots)$ grows as e_i grows. In the preceeding example, the constructor *pair* is positive in both positions. This is the meaning behind the `+` syntax in ibanshee: it allows the user to tell IBANSHEE that a constructor has positive variance in some position.

The correct variance depends on what the constructor is intended to model. Let's look at another example:

Example 5 Suppose we have a binary constructor fun that models sets of functions (the first argument in the function domain, the second argument the function range). Suppose we also have sets *int* containing the integers, and *real* containing the real numbers, with $int \subset real$. Now consider the sets $fun(int, int)$ and $fun(real, int)$. Which set is larger? Consider any function in the set $fun(real, int)$. Such a function expects a *real* argument. However, any *int* is also a *real*, so any function in $fun(real, int)$ is also in the set $fun(int, int)$. Therefore, $fun(real, int) \subseteq fun(int, int)$. In this case, if e_1 shrinks, $fun(e_1, e_2)$ grows.

We say that a constructor c has *negative* variance in its ith position if the set $c(\ldots, e_i, \ldots)$ shrinks as e_i grows. In the preceeding example, the constructor fun is negative in its first argument. In IBANSHEE, this can be specified by preceeding a sort name with a `-` sign:

```
[0] > fun(-setIF,+setIF):setIF
constructor: fun
```

With a little thought, it's easy to see that the range of a function should have positive variance.

Another way of looking at variances is as follows: if $c(\ldots, e_i, \ldots) \subseteq c(\ldots, e'_i, \ldots)$, with c positive in i, then e'_i contains *at least* the set e_i ($e_i \subseteq e'_i$). If c is negative in i, then e'_i contains *at most* the set e_i ($e'_i \subseteq e_i$). There is actually a third kind of variance, *non* variance. If c is non variant in i, then e'_i contains *exactly* the set e_i ($e_i = e'_i$). In IBANSHEE, you get non variance by using `=` in place of `+` or `-`:

```
[0] > ref(+setIF,=setIF):setIF
constructor: ref
```

3.6 Sorts

A BANSHEE sort is a combination of

- a language of expressions,
- a constraint relation between expressions,
- a solution space,
- and a resolution algorithm

Here we give a brief description of the sorts available in BANSHEE.

3.6.1 Sets in Inductive Form

Up to this point, we've restricted ourselves to `setIF` expressions. The `IF` stands for *inductive form*, which is a way of representing and solving set constraints efficiently [1]. The `setIF` sort provides inclusion constraints between set expressions. The solution space is sets of regular trees.

3.6.2 Sets in Subtransitive Form

BANSHEE also supports subtransitive form, an alternative algorithm for solving set constraints [5]. Other than having a different resolution algorithm, `setST` is identical to `setIF`.

3.6.3 Terms

The `term` sort provides equality constraints between expressions. The solution space is regular trees. The resolution algorithm is essentially Robinson's unification algorithm.

The `term` sort also provides conditional equality constraints [8]. A conditional equality constraint $t_1 \leq t_2$ is satisfied if either $t_1 = 0$, or $t_1 = t_2$.

3.6.4 Rows

A *row* is a finite map from strings to expressions. For each sort s, BANSHEE provides a corresponding row sort $row(s)$ that maps strings to expressions of sort s. Rows essentially model *records* with any combination of width and depth subtyping. Note that BANSHEE does not support rows of rows.

3.7 Mixed Constraints

One of the most powerful features of BANSHEE is its support for *mixed constraints*. Mixed constraints permit constructors to contain subexpressions of different sorts. This allows the user to find the "sweet spot" between efficiency and precision when designing an analysis. Mixed constraints are easy to use: simply define a constructor that uses multiple sorts:

```
[0] > f(+setIF,=term):setIF
constructor: ref
```

With `f` defined as above, it is possible to add constraints between mixed expressions (expressions with subexpressions of different sorts):

```
[0] > 'x : setIF
[0] > 'y : term
[0] > 'z : setIF
[0] > c : term
[0] > f('x, c) <= f('z,'y)
[1] > !ecr 'y
c
```

3.8 Backtracking

BANSHEE has a feature called *backtracking* that allows the user to "roll back" the state of a constraint system at any point in time. In IBANSHEE, the number displayed at the prompt tells the current "version" of the constraint system. Backtracking allows the constraint system to be rolled back to any previous version, by specifying the version number:

Let's work with the previous example:

```
[0] > f(+setIF,=term):setIF
constructor: ref
[0] > 'x : setIF
[0] > 'y : term
[0] > 'z : setIF
[0] > c : term
[0] > f('x, c) <= f('z,'y)
[1] > !ecr 'y
c
```

Notice that the version number is incremented after the addition of the constraint `f('x,c) <= f('z,'y)`. That means that the constraint system's version prior to the addition of that constraint is `0`, and the version after the constraint addition is `1`. In IBanshee, backtracking is accomplished by the command `!undo [i]`, where `i` is the version of the constraint system to backtrack to:

```
[1] > !undo 0
[0] > !ecr 'y
'y
```

After the `!undo` command, the constraint system reverts to its state just before the constraint was added. Since `'y` is unconstrained, its equivalence class representative is itself.

3.9 Persistence

Constraint systems can be saved or loaded to disk. The complete internal representation of a constraint system is saved, so after restoration all operations are legal (including backtracking).

4 Using IBanshee

4.1 IBanshee Commands

- `!help` Print the quick reference
- `!tlb e` Print the transitive lower bounds of `e`
- `!ecr e` Print the equivalence class representative of `e`
- `!undo i` Roll back the constraint system to its state at time `i`
- `!trace i` Set the trace level to depth `i`. This will print constraints for recursive calls to the constraint solver up to `i` levels deep. Helpful for debugging.
- `!quit` Exits iBanshee
- `!save "filename"` Saves the current constraint system
- `!load "filename"` Loads the constraint system saved using the `save` command
- `!rsave` Saves the current constraint system (the filename is currently hard-coded) using region serialization. This is much faster than using `!save`.
- `!rload` Load the constraint system saved with `rsave`.
- `!exit` Exits iBanshee

5 Using the Nonspecialized Engine API

In addition to the interactive constraint solver IBanshee, Banshee includes a nonspecialized C library (`[engine/libnsengine.a]`) that provides a direct interface to the constraint solver.

To use Banshee as a nonspecialized library, these steps should be followed:

- The analysis designer should include the header file `[engine/nonspec.h]` in their application. In general, this should be the only Banshee header included.
- The client should call `nonspec_init` before calling any other Banshee functions.
- The client should be linked with the libraries `[engine/libnsengine.a]` and `[libcompat/libregions.a]`.

The header `[engine/nonspec.h]` contains the complete nonspecialized Banshee API.

6 Using the Engine Generator

One of the major innovations in Banshee is *specialization*. Banshee includes a code generator that specializes the constraint back-end for each program analysis. The analysis designer describes a set of constructor signatures in a specification file, which Banshee compiles into a specialized constraint resolution engine. Specialization allows checking a host of correctness conditions statically, and also improves software maintenence. The insight here is that most program analyses only use a fixed, static set of non-constant constructors. The Banshee code generator essentially does partial evaluation of the constraint solver with respect to the statically defined constructor signatures.

To use Banshee as a nonspecialized library, these steps should be followed:

- The analysis designer writes a short *specification* describing the signatures of constructors used in the analysis. Other than constants, only the constructors described in the specification will be available (new nonconstant constructors cannot be definedd dynamically).
- The analysis designer runs the code generator `[bin/banshee.exe]` on their specification file. This generates a C source file and header file, containing the interfaces for the new specialized engine.
- The analysis designer includes the generated header file (and ideally no other Banshee headers) in the client application
- The client application calls the appropriate initialization routine (a function whose name ends with `_init`) before calling any other Banshee function

- The analysis designer links their client with `[engine/libengine.a]`, the .o file produced by compiling the generated C source file, and the region library `[libcompat/libregions.a]`.

6.1 Banshee Specifications

Banshee specifications contain the same information as IBanshee constructor declarations, but the definitions are introduced statically, before any constraints or expressions are built. We illustrate the syntax with a simple example (from `[lambda/lambda.bsp]`

```
specification lambda : LAMBDA =
 spec
  data l_type : term = boolean
     | integer
             | function of l_type * l_type
end
```

In IBanshee syntax, this specification would be roughly identical to the following constructor definitions:

```
[0] > boolean : term
constructor: boolean
[0] > integer : term
constructor: integer
[0] > function(=term,=term) : term
constructor: function
```

Note that the Banshee specification file is slightly more fine-grained: the IBanshee constructor defintions allow *any* expression of sort `term` to appear as subexpressions to the `function` constructor. However, in the specification file, only `l_type` expressions can appear in the `function` constructor. Conceptually, each `data` declaration defines a new "type", whereas in IBanshee (and the nonspecialized library) each sort is a separate type.

Using a specialized engine has a number of advantages to using the nonspecialized engine. Unless the client analysis requires a dynamic set of non-constant constructors, we recommend specialization.

See `[tests/lambda.bsp]`, `[tests/lambda-test.c]` and the associated `make` target in the `tests` directory for a complete example.

7 Using the Dyck CFL Reachability API

Many program analyses can be reduced to a context free language reachability problem over languages of matched parentheses (the *Dyck* languages). In [6], we describe an efficient reduction from set constraints to Dyck CFL reachability. We have implemented this reduction and have included it with Banshee.

The API for Dyck CFL reachability using BANSHEE can be found in `[dyckcfl/dyckcfl.h]`. To use this API, the following steps should be taken:

- The analysis designer should include the headers `[engine/nonspec.h]` and `[dyckcfl/dyckcfl.h]`, and ideally, no other BANSHEE headers.
- Functions `nonspec_init` and `dyck_init` should be called first.
- The analysis designer should link the libraries `[engine/libnsengine.a]` and `[libcompat/libregions.a]` along with the object file `[dyckcfl/dyckcfl.o]`.

See `[tests/dyckcfl-test.c]` and the associated `make` target in the `tests` directory for a complete example.

8 Reference

8.1 IBanshee Quick Reference

```
ident : [A-Z a-z]([A-Z a-z 0-9 _])* integers (i) : [0-9]+

Variables (v)    : '{ident}
Constructors (c) : {ident}
Labels (l)       : {ident}
Names (n)        : {ident}

Expressions (e)  : v | c | n | c(e1,...,en) | e1 && e2 | e1 || e2
                 | <l1=e1,...,ln=en [| e]> | 0:s | 1:s | _:s
                 | pat(c,i,e) | proj(c,i,e) | ( e ) | c^-i(e)

sorts            : basesort | row(basesort)

basesort         : setIF | term

Var decl         : v : sort
Constructor decl : c(s1,...,sn) : basesort
Name decl        : n = e
Sig (s)          : + sort | - sort | = sort

Constraints      : e1 <= e2 | e1 == e2

Commands         : !help
                   !tlb e
                   !ecr e
                   !undo [i]
                   !trace [i]
                   !quit
```

```
!save "filename"
!load "filename"
!rsave
!rload
!exit
```

8.2 Grammar for Banshee Specifications

Specification files consist of a single *specification*, drawn from the following grammar:

specification	::=	`specification` *spcid* `:` *hdrid* `=` `spec` *dataspec* `end`
dataspec	::=	`data` $exprid_1$ `:` $sort_1\langle sortopts\rangle$ $\langle$`=` $conspec_1\rangle_1 \cdots$
		`and` $exprid_n$ `:` $sort_n\langle sortopts\rangle$ $\langle$`=` $conspec_n\rangle_n$
	|	$dataspec_1$ $dataspec_2$
conspec	::=	*conid* $\langle$`of` *consig*$\rangle$
	|	$conspec_1$ `|` $conspec_2$
consig	::=	$bconsig_0$ `*` $\cdots$ `*` $bconsig_n$
bconsig	::=	*vnc exprid*
vnc	::=	`+`|`-`|`=`
sortopts	::=	`[`$option_1$`,` $\cdots$ `,`$option_n$`]`
sort	::=	`term` | `setIF` | `setST` | `row(`*exprid*`)`

Expression identifiers (*exprid*), constructor identifiers (*conid*), specification identifiers (*spcid*), and header identifiers (*hdrid*) must only be used once in a `.spec` file. Identifiers consist of an upper- or lowercase letter followed by a sequence of letters, digits, or underscores ('_').

References

[1] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, 1998.

[2] M. Fähndrich. *BANE: A Library for Scalable Constraint-Based Program Analysis*. PhD dissertation, University of California, Berkeley, Department of Computer Science, 1999.

[3] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 114–126, London, United Kingdom, 1997. Springer-Verlag.

[4] N. Heintze. *Set Based Program Analysis*. PhD dissertation, Carnegie Mellon University, Department of Computer Science, Oct. 1992.

[5] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[6] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 207–218. ACM Press, 2004.

[7] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS '05: Proceedings of the 12th International Static Analysis Symposium*. London, United Kingdom, September 2005.

[8] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.

A Known Bugs and Limitations

- If backspace doesn't delete properly in IBANSHEE, try typing `stty erase ^?` at the shell prompt before invoking ibanshee.exe.
- Projection merging and the DyckCFL API don't work properly (the bug has to do with group projections and merging). TURN OFF projection merging if using the DyckCFL API with globals. If you're not using globals, this won't matter, because group projections won't be used. Projection merging doesn't improve the DyckCFL library's performance, anyway.
- Hashset (`[hashset.c]`) sometimes doesn't terminate. By default, banshee doesn't use hashset: it uses a bounds representation based on `[hash.c]` (`[hash_bounds.c]`, as opposed to the hashset based `[bounds.c]`). However, the hashset-based bounds representation is about twice as fast as the hash-based bounds representation. If you want to try out the hashset-based bounds representation, do `make clean all check NO_HASH_BOUNDS=1`. Note that backtracking hasn't been implemented in hashset, so some of the check targets will fail.
- Andersen's analysis requires pre-processed source files. We recommend pre-processing with gcc-2.95, as pre-processing with later versions of GCC may produce output that our C front-end cannot parse. Thanks to Stephen McCamant for this suggestion.

B Copyright

- `[engine/malloc.c]`: see file for license
- `[codegen/OCamlMakefile]`: see http://www.ai.univie.ac.at/ markus/home/ocaml_sources.html for license
- `[cparser/*]`: GPL, see cparser/COPYRIGHT. Note that the cparser is a separate application, no code from cparser is linked into BANSHEE itself.

See file `[COPYRIGHT]` for more details.