Compiling with continuations

Vincent St-Amour

March 26, 2010

1 ORBIT: an optimizing compiler for scheme

```
@article{orbit,
 author = {Adams, Norman and Kranz, David and Kelsey, Richard and
           Rees, Jonathan and Hudak, Paul and Philbin, James},
 title = {ORBIT: an optimizing compiler for scheme},
 journal = {SIGPLAN Not.},
 volume = {21},
 number = {7},
 year = {1986},
 issn = {0362-1340},
 pages = {219--233},
 doi = {http://doi.acm.org/10.1145/13310.13333},
 publisher = {ACM},
 address = {New York, NY, USA},
}
```

1.1 Content

This paper describes the Orbit Scheme optimizing compiler. Orbit was one of the first practical compilers for Scheme, whereas most existing Scheme compilers at the time were research prototypes. The paper walks us through the phases of the compiler, explaining design decisions along the way. The result of this research is a production-quality Scheme compiler. It generates code that is competitive with the best LISP compilers of the day, as shown in the benchmarks present in the paper.

1.2 Ideas

The design of Orbit reuses the idea, first introduced by Steele's Rabbit compiler, of using continuation passing style as a compiler's intermediate representation. It also significantly improves on the technique by developing several optimizations operating on CPS code such as assignment conversion and trace scheduling. In addition, the paper also presents several efficient closure implementations, each applicable in different situations. Orbit established CPS as a credible internal representation for compiling funcitonal languages.

2 Continuation-passing, closure-passing style

```
@inproceedings{cpscps,
 author = {Appel, A. W. and Jim, T.},
 title = {Continuation-passing, closure-passing style},
 booktitle = {POPL ’89: Proceedings of the 16th ACM SIGPLAN-SIGACT
              symposium on Principles of programming languages},
 year = {1989},
 isbn = {0-89791-294-2},
 pages = {293--302},
 location = {Austin, Texas, United States},
 doi = {http://doi.acm.org/10.1145/75277.75303},
 publisher = {ACM},
 address = {New York, NY, USA},
}
```

2.1 Content

This paper presents the CPS code generator implemented in the Standard ML of New Jersey compiler, which led to impressive speedups compared to its original stack-based code generator. The paper goes over each phase of the new back-end and present benchmarks comparing it to their old one, and to the Orbit and Berkeley Pascal compilers.

2.2 Ideas

Built to improve the performance of SML/NJ, this new back-end borrows CPS-based optimization techniques developed in Orbit. This paper (and several others) shows that these techniques apply not only to Scheme-like languages, but that they are also valid and useful for languages of the ML family (sound static typing, complex module system, pattern matching, etc.). The authors also improve on Orbit’s techniques by using ML’s type system to ensure the well-formedness of CPS expressions. Furthermore, the paper names some techniques introduced by Orbit that had not been recognized as independent techniques before, such as closure conversion. This clean separation and naming of phases made it possible to study them individually, which has often been done since. For example, several papers study closure conversion by itself.

3 The essence of compiling with continuations

```
@inproceedings{essence,
 author = {Flanagan, Cormac and Sabry, Amr and Duba, Bruce F.
           and Felleisen, Matthias},
 title = {The essence of compiling with continuations},
 booktitle = {PLDI ’93: Proceedings of the ACM SIGPLAN 1993
              conference on Programming language design and
              implementation},
 year = {1993},
 isbn = {0-89791-598-4},
 pages = {237--247},
 location = {Albuquerque, New Mexico, United States},
 doi = {http://doi.acm.org/10.1145/155090.155113},
 publisher = {ACM},
 address = {New York, NY, USA},
}
```

3.1 Content

This paper exposes issues with the use of CPS as a compiler IR: naïve CPS conversion introduces administrative λ-expressions that increase the size of the program, and code generation inverts CPS translation. The authors propose the ANF (administrative normal form) as an alternative to CPS. They then go on to argue that the ANF term derived from a given program is equivalent to the term obtained by converting the program to CPS, simplifying the result, and then undoing CPS. The authors also provide a linear-time A-normalization algorithm.

3.2 Ideas

The authors noticed that, to solve the issues with CPS outlined above, realistic CPS compilers implemented various ad-hoc techniques. The authors observed similarities between these techniques and formalized them by introducing ANF. Notably, ANF makes a distinction between continuation closures (introduced by CPS) and source-language closures. It is a useful distinction that has an important effect on the efficiency of closures, as explained in the paper on Orbit. However, that distinction is not explicit in CPS, and most realistic CPS compilers ended up extracting it through various analyses. Even though ANF, unlike CPS, cannot use the full λ-calculus for optimizations, the $lambda_v$-calculus that it can use is rich enough to do all the transformations that a compiler would do on CPS code at the time.

4 Local CPS conversion in a direct-style compiler

```
@inproceedings{localcps,
    author = "John Reppy",
    title = "Local {CPS} conversion in a direct-style compiler",
    booktitle = "Proceedings of the {Third} {ACM} {SIGPLAN}
                 {Workshop} on {Continuations} ({CW}'01)",
    pages = "13--22",
    year = "2001",
    url = "citeseer.ist.psu.edu/reppy01local.html"
}
```

4.1 Content

This paper presents a new compiler transformation, local CPS conversion, that brings the advantages of using CPS to the parts of the program that would benefit from them while still having a direct style compiler. The author motivate this transformation by showing that CPS opens up some interesting loop optimizations that a direct style IR cannot do. He then goes over an analysis that determines when it is useful to apply the transformation and explains the transformation itself. These ideas have been implemented in the Moby compiler.

4.2 Ideas

While ANF and related direct style IRs preserve most of the advantages of compiling with continuations (simple register allocation and code generation, full λ-calculus for optimization, etc.), they are not suitable for some optimizations, notably loop optimizations. To solve this problem, local CPS conversion is introduced. By having portions of the program in CPS, more optimizations can be performed, including some traditional SSA[1]-based optimizations.

[1]see: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, by Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck (1991)